Error Detection for Soft Computing Applications

by

Anna Thomas

B.Tech., National Institute of Technology Calicut, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)
August 2013

© Anna Thomas 2013
Abstract

Hardware errors are on the rise with reducing chip sizes, and power constraints have necessitated the involvement of software in hardware error detection. At the same time, emerging workloads in the form of soft computing applications, (e.g., multimedia applications) can tolerate most hardware errors as long as the erroneous outputs do not deviate significantly from error-free outcomes. We term outcomes that deviate significantly from the error-free outcomes as Egregious Data Corruptions (EDCs). In this thesis, we propose a technique to place detectors for selectively detecting EDC causing errors in an application. Our technique identifies program locations for placing high coverage detectors for EDCs using static analysis and runtime profiling. We evaluate our technique on six benchmarks to measure the EDC coverage under given performance overhead bounds. Our technique achieves an average EDC coverage of 82%, under performance overheads of 10%, while detecting only 10% of the Non-EDC and benign faults. We also explore the performance-resilience tradeoff space, by studying the effect of compiler optimizations on the error resilience of soft computing applications, both with and without our technique.
Preface

This thesis is based on a work conducted by myself in collaboration with Dr. Karthik Pattabiraman. The work was published as a conference paper in the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) [39]. I was responsible for coming up with the solution and validating it, evaluating the solution and analyzing the results, and writing the paper. Karthik was responsible for guiding me with the solution reasoning, experiments design and results analysis, as well as editing and writing portions of the paper.

Anna Thomas and Karthik Pattabiraman, Error Detector Placement for Soft Computation, the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks, 2013

Parts of this thesis is based on the following workshop papers [38, 40]:


# Table of Contents

Abstract ................................................................. ii

Preface ................................................................. iii

Table of Contents ..................................................... iv

List of Tables .......................................................... viii

List of Figures ........................................................ ix

List of Acronyms ....................................................... xi

Acknowledgements ....................................................... xii

Dedication .............................................................. xiii

1 Introduction .......................................................... 1
   1.1 Motivation ....................................................... 1
   1.2 Research Goal and Proposed Solution ......................... 3
   1.3 Contributions ................................................... 4
   1.4 Background ...................................................... 6
   1.5 Related Work .................................................... 9
# Table of Contents

1.6 Thesis Organization ............................................. 14
1.7 Summary ......................................................... 14

2 LLFI ................................................................. 16
  2.1 Motivation ..................................................... 16
  2.2 Methodology .................................................. 17
  2.3 Accuracy of LLFI ............................................. 20
  2.4 Implementation .............................................. 23
  2.5 Results on LLFI Accuracy ................................. 24
  2.6 Summary ......................................................... 26

3 Preliminary Study .................................................. 27
  3.1 Fault Model ................................................... 27
  3.2 Fault Injection Experiment ................................. 28
  3.3 Fault Injection Results ...................................... 31
  3.4 Summary ......................................................... 33

4 Approach ............................................................ 34
  4.1 Assumptions about Soft Computation ..................... 34
  4.2 Heuristics ....................................................... 35
    4.2.1 Step 1: Function Identification ....................... 38
    4.2.2 Step 2: Data Categorization ......................... 40
  4.3 Approach ....................................................... 43
    4.3.1 EDC Ranking Algorithm ................................. 44
    4.3.2 Selection Algorithm ..................................... 48
  4.4 Summary ......................................................... 52
# Table of Contents

5 Experimental Setup ........................................... 53  
5.1 Implementation ........................................... 53  
5.2 Benchmarks ................................................ 55  
5.3 Coverage Evaluation ...................................... 58  
5.4 Summary .................................................... 59  

6 Results ....................................................... 60  
6.1 Error Outcome Rates ...................................... 60  
6.2 Coverage Under Varying Performance Overheads ........ 61  
6.3 EDC Coverage under varying funcrange values ........... 64  
6.4 EDC coverage under varying fidelity threshold .......... 65  
6.5 Quantitative comparison with Prior Work ................ 66  
6.6 Discussion .................................................. 68  
6.7 Summary .................................................... 69  

7 Effect of Compiler Optimizations ........................... 70  
7.1 Compiler Optimizations ................................... 71  
7.1.1 Dichotomous Behaviour on Baseline Resilience ...... 71  
7.1.2 Change in heuristics .................................... 73  
7.1.3 Change in EDC-prone code regions ................. 75  
7.2 Methodology ............................................... 76  
7.3 Results ..................................................... 78  
7.3.1 Effect of compiler optimizations on Baseline Resilience 78  
7.3.2 Effect of compiler optimizations on detection tech-  

7.4 Overall Vulnerability ...................................... 88
Table of Contents

7.5 Summary .............................................. 93

8 Conclusion and Future Work .......................... 94
  8.1 Conclusion ......................................... 94
  8.2 Future Work ....................................... 95

Bibliography .............................................. 97
List of Tables

2.1 Difference between LLVM Intermediate Code and Assembly Language ........................................ 21

3.1 Classification of faults according to data categories .............. 30

4.1 Attribute Values and Static EDC Rank for data items using an example ........................................ 45

5.1 Characteristics of Benchmark Programs .............................. 56

6.1 Percentage of Error outcomes in each benchmark ............... 61

7.1 Percentage of EDC outcomes in each benchmark under the fourteen optimizations versus unoptimized version ............... 79

7.2 The overall vulnerability of the fourteen optimizations both with and without our technique ....................... 91
List of Figures

1.1 An image classified as an EDC versus a Non-EDC image . . . 7
1.2 Example Code of EDC versus Non-EDC data . . . . . . . . 8

2.1 C code for factorial program . . . . . . . . . . . . . . . . . . 19
2.2 LLVM IR for the factorial program . . . . . . . . . . . . . . 19
2.3 LLVM IR of Figure 2.2 instrumented with fault injection calls 19
2.4 Comparison of fault injection outcome percentages between 25
    LLFI and PIN fault injector . . . . . . . . . . . . . . . . . .

3.1 Fault-Injection outcomes as per different data categories . . 31

4.1 Example Code for Function and Data Categorization . . . . 37
4.2 Technique Workflow with required inputs . . . . . . . . . . 43
4.3 Pseudo-code to show the calculation of weighted rank . . . 51

6.1 EDC Coverage using our technique under performance over- 62
    heads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
6.2 Non-EDC and Benign Coverage for our technique under vary- 63
    ing performance overheads . . . . . . . . . . . . . . . . . .
6.3 Effect of varying funcrange on EDC Coverage . . . . . . . 64
List of Figures

6.4 Average EDC coverage under varying fidelity thresholds . . . 66
6.5 EDC Coverage by Selective Duplication Technique . . . . . . 67

7.1 Loop Invariant Code Examples in Swaptions . . . . . . . . . 82
7.2 Blackscholes code segment . . . . . . . . . . . . . . . . . . . . 85
7.3 Normalized EDC Coverage of different optimizations with re-
spect to the Unopt EDC Coverage (shown as 100%). . . . . . 88
List of Acronyms

LLVM  Low Level Virtual Machine
IR    Intermediate Representation
LLFI  LLVM Based Fault Injector
MTTU  Mean Time To Upset
SDC   Silent Data Corruption
EDC   Egregious Data Corruption
SSA   Static Single Assignment
JIT   Just-In-Time
I would like to thank my advisor Dr. Karthik Pattabiraman for all the guidance and support he has offered me during my graduate study. Karthik has been very instrumental in moulding my approach to research problems and encouraging me to come up with solutions. I have had some of the most insightful discussions during the initial months of formulating my research problem. His consistence in research and patience in dealing with problems is something I find very inspiring.

I also take this opportunity to thank my labmates who made the lab quite an enjoyable place to work, with the discussions about research and life in general. Thanks to my colleagues and the professors at the CSRG meetings who helped me think about my research work from a broader perspective, and offered valuable feedback during practice talks.

I thank my wonderful friends here and back home, for helping me take things lightly. Special thanks to my friends across the border, Ankit and Preeti, for all the fun talks and advice on critical decisions. Finally, I thank my family for the emotional support they provided. My constant source of joy was explaining to them in the simplest terms, the problem I was working on. This would not have been possible without their support.
Dedication

To my parents
Chapter 1

Introduction

1.1 Motivation

With increasing transistor densities and reducing feature sizes, chips must tolerate larger global and local process variations [1], and a reduction in the Mean-Time-To-Upset (MTTU) [2]. Moreover, microprocessors are expected to experience significantly higher rates of hardware faults in the near future because of the decrease in manufacturing process and voltage [4]. For instance, studies have shown that alpha-induced transient faults increase 30 times when the manufacturing process goes from 250\text{nm} to 180\text{nm} and the voltage drops from 2\text{V} to 1.6\text{V} [16].

Traditionally, hardware errors have been tolerated through hardware redundancy (e.g. Dual Modular Redundancy) or guard banding. We motivate the need to provide fault-tolerance in software for hardware errors by two observations: (1) The hardware-only solutions have high energy and performance overheads. These solutions become untenable as power consumption becomes a dominant concern in processor design [6]. (2) Prior work has shown that a large number of faults are masked when propagating to higher levels of the system stack such as microarchitecture, architecture and application levels [11, 29, 42]. We only need to protect against those faults that
1.1. Motivation

propagate through the hardware layers and are visible at the application layer, thus avoiding wasteful detection and recovery.

Recently, there have been several proposals to selectively expose hardware faults to the software layer and tolerate them [5, 12, 24, 26, 30]. These proposals leverage the ability of certain software applications to tolerate faults in their data, and still produce acceptable outputs. Such applications are called soft computing applications [44]. Soft computing applications have gained increasing prominence, and researchers have predicted that future workloads will belong primarily to this category [14].

Examples of soft computing applications are multimedia decoding applications, which can tolerate blurry decoded images, and machine learning applications, which can tolerate noise. These applications have an associated fidelity metric, which is a quantitative measure of the output quality. For example, in the case of image and video decoders, the fidelity metric is peak signal-to-noise ratio (PSNR). As long as the produced output quality does not deviate significantly from the fidelity metric, it is deemed acceptable. We use the term *Egregious Data Corruptions (EDCs)* to denote outcomes that deviate significantly from the fidelity metric, i.e., unacceptable outcomes.

The error tolerance of soft computing applications does not mean that they are resilient to all errors. In particular, an error in a soft computing application may or may not lead to an EDC. If it will lead to an EDC, then the application needs to be stopped, as otherwise, its output will be unacceptable. On the other hand, if the error will not lead to an EDC, it is better to let the application continue rather than perform wasteful detection and recovery, and incur unnecessary overheads. This overhead will become
more prominent as error rates increase, as they are predicted to do so in future processors.

1.2 Research Goal and Proposed Solution

We address the following research question in our work: *Is there a systematic technique to place error detectors in soft computing applications for EDC causing errors?*

Our goal is to efficiently place error detectors in soft-computing applications in order to detect errors early (thus avoiding egregious outcomes), at the same time detecting only those errors that lead to EDCs (thus avoiding wasteful recovery). An error detector is an assertion or check on one or more data variables in the application. We develop heuristics that determine where to place error detectors for avoiding EDCs, using fault injection and static analysis of the application’s code. While we use fault injection to develop the heuristics, we do not require fault injection to apply the developed heuristics to new applications. This is because our heuristics are based on static and dynamic properties of the application’s code, and do not rely on semantic knowledge of the application. Note that fault-injection is a time intensive process for large applications, and hence it is desirable to avoid it (if possible).

Prior work [18, 22, 32] has investigated the problem of optimal error detector placement. However, these techniques focus on placing detectors to minimize the error detection latency or to detect specific failures such as safety violations. In particular, they do not consider optimizing the detector
placement for minimizing the EDC rate, which is important for soft com-
puting applications. As we show later in this thesis, minimizing the EDC rate leads us to different placement decisions than if we had optimized for minimizing the number of Silent Data Corruptions (SDCs), which constitute any deviation from the correct output (not only egregious ones) [17].

In this work, we develop a detector placement technique to improve the resilience of soft computing applications. Resilience (R) is the property of the application to prevent an error from becoming an EDC, given that the error has already occurred in the application. Higher resilience is better. In mathematical terms, this is defined as

\[ R = P(\text{fault} \neq \text{EDC} \mid \text{fault occurred in app}) \]  

(1.1)

This is different from vulnerability (V), which is the likelihood of a fault affecting the application and the fault leading to an EDC. A lower vulnerability is better. Vulnerability can be defined in terms of resilience (see Equation (1.2)).

\[ V = P(\text{fault occurred in app}) \times P(\text{fault} = \text{EDC} \mid \text{fault occurred in app}) \]

\[ = P(\text{fault occurred in app}) \times (1 - R) \]  

(1.2)

1.3 Contributions

We make the following contributions in this thesis:

1. We build and validate LLFI, a compiler-based fault injector. LLFI
allows fault-injections to be performed into specific data types at the intermediate code level of the LLVM compiler, and to trace the propagation of the fault in the program. We also validate the accuracy of LLFI compared to fault injection at the assembly level, for soft computing applications (Chapter 2).

2. We perform fault injection into soft computing applications using LLFI, and distinguish EDCs from the set of SDCs. Based on the injections, we develop heuristics for identifying EDC-prone regions of code or data, which are appropriate candidates for detector placement (Chapter 3).

3. We develop a systematic algorithm based on these heuristics, that (a) ranks the data according to their EDC causing nature, based on static analysis (b) uses a greedy approach that combines the static information, with the dynamic execution profile, to choose the appropriate set of EDC data or code for placing detectors. Our algorithm takes as input the application source code, the acceptable performance overhead, and the execution profile data of the application, and identifies the locations to place detectors in the program, i.e., data or instructions (Chapter 4).

4. We implement the algorithm within the LLVM compiler, and evaluate its accuracy through fault-injection. We find that the detectors placed by the algorithm provide EDC coverage of 82% under 10% performance overhead, while providing a Non-EDC and benign coverage of only 10% (Chapter 6).
5. We study the effect of individual compiler optimizations on the error resilience of soft computing applications. We focus on how these optimizations affect the resilience of these applications, both with and without our technique. While compiler optimizations may increase application performance, they might lower the application error resilience (Chapter 7).

1.4 Background

Egregious Data Corruptions (EDCs) are application outcomes that deviate significantly from the fault free outcome, i.e., they affect outputs egregiously. This deviation is quantified by a fidelity metric that is well defined for most soft computing applications [25]. For example, the fidelity metric used by speech decoders is Segmental SNR. Silent Data Corruptions (SDCs), or outcomes that result in any deviation in the output from the fault free outcome, are a superset of EDCs. An SDC is classified as EDC or Non-EDC, depending on the fidelity threshold value of the outcome. Non-EDCs are the SDCs with small deviation in output, i.e., SDCs that do not belong to the category of EDCs.

EDC is a relative term as it depends on how the user sets the fidelity threshold. In this work, we focus on detecting errors under the assumption that the user tolerates most small deviations in outputs, i.e., the application is used under relaxed conditions. For example, in image and video decoding applications, we set the fidelity threshold based on whether the frames are corrupted to the point of being unrecognizable or are of very poor image
1.4. Background

quality. In other cases, where we cannot rely on human perception, we set the fidelity threshold to be such that around 30% of the most egregious SDCs are categorized as EDCs (see Chapter 5).

![Figure 1.1: The EDC causing fault decoded image (left) versus Non-EDC causing fault decoded image (right) from the JPEG decoder](image)

The example in Figure 1.1 shows the faulty decoded images of the JPEG decoder (part of Mediabench [21]), when a fault is injected into the program. The fidelity threshold is Peak Signal to Noise Ratio (PSNR) between the fault-free decoded image, and the faulty decoded image. As the PSNR value becomes lower, the output corruption becomes more egregious. Assuming a fidelity threshold value of PSNR 30, the faulty image on the left with a PSNR of 11.37 is classified as an EDC, while the faulty image on the right with a PSNR value of 44.79 is classified as a Non-EDC. The comparison is performed with respect to the base image, which we do not show.

**Example:** We now explain the correlation between an EDC and the program characteristics, using a function \texttt{conv422to444} from the example code shown in Figure 1.2. This example is based on the MPEG benchmark, from the Mediabench Benchmark [21]. The function \texttt{conv422to444} converts from YUV 4:2:2 subsampling (U and V components are sampled at half the
1.4. Background

rate of $Y$ component) to YUV 4:4:4 (all components sampled at same rate).

```c
void conv422to444(char *src, char *dst, int width, int height, int offset)
{
    ...
    w = width >> 1;
    if(dst < src + offset) //B1
        return;
    ...
    for (j=0; j < height; j++) { //B2
        for (i=0; i < width; i++) { //B3
            i2 = i<<1;
            im1 = (i < 1) ? 0 : i-1; //B4
            ...
            dst[i2] = Clip[(21*src[im1]) >> 8]; //P1
        }
        if(j + 1 < height) { //B5
            src += w; //P2
            dst += width;
        }
    }
    ...
}
```

Figure 1.2: Example Code of EDC versus Non-EDC data

**Challenge:** Prior research has shown that faults in data values of code constituting higher dynamic execution time are more likely to cause SDCs [17]. In other words, SDC causing code tends to be on the hot paths of the application. However, EDCs are caused by a large deviation in output, and are not necessarily caused by faults in data on the hot paths. For example, in Figure 1.2, the longest running statements are the lines 9 to 12,
the ones within two nested `for` loops. A fault at the branch `i < 1` at B4, or at the pointer data P1, causes an SDC but not an EDC. However, a fault occurring at loop termination conditions B2 and B3 cause an EDC.

Therefore, to maximize the coverage for EDCs, detectors should be placed at code regions or data that have the highest impact on the application’s output. The main challenge in detecting EDCs is coming up with a general algorithm to identify such code or data. Further, the algorithm should be based only on the static code of the program and its execution profile, and not require fault injections, which are expensive. This is the main challenge we address in our work.

### 1.5 Related Work

We classify related work into three areas, namely (1) identifying critical variables, (2) placing error detectors in a program, and (3) effect of compiler optimizations on application vulnerability.

**Identifying Critical Variables** A critical variable is defined as a variable that would cause a particular outcome (e.g., SDCs), when a fault occurs at that variable. There has been significant work on identifying critical variables for different kinds of failure outcomes in applications.

Cong and Gururaj [8] focus on identifying all critical variables in an application that can tolerate deviations in output (i.e., soft computing applications). Similar to our work, they also consider outcomes that cause large deviations from the correct output i.e., EDCs. They develop an algorithm to identify critical variables using static analysis, runtime profiling, and a
1.5. Related Work

runtime monitoring mechanism. Their approach differs from our work in two ways. First, they consider protecting critical variables, rather than placing detectors. As a result, they can incur much higher overheads than our technique, because protecting critical variables often involves duplicating the hot paths of the application. Second, their technique uses two versions of code - full duplication to ensure numerically accurate outputs, versus selective replication of only the critical variables. Based on the decision taken by the runtime monitoring mechanism, their solution switches to full duplication. Further, they do not present the EDC rates and the EDC coverage of the benchmark applications, which makes it difficult to quantitatively compare their technique with ours.

Rehman et al. [33] identify the set of critical instructions to protect under a user specified performance overhead bound. They use mathematical models to quantify the vulnerability of different types of instructions by calculating the fault rate and error masking probabilities. Based on the function resilience, and instruction vulnerability and error masking probabilities, they provide more protection to less resilient instructions under a given performance overhead bound. This is similar to our work of selectively detecting EDC causing faults. However, their work differs from ours in the following aspects: (1) they focus on all faults that lead to incorrect outputs, rather than those that lead to EDCs. Further, they consider the vulnerability of individual instructions, i.e., the unconditional probability that a fault affects a particular instruction, by incorporating fault rates. This requires detailed fault injection campaigns to estimate the parameters required in their models. Since we focus on resilience, which is a property of the ap-
application alone, we do not require fault injections to apply our technique to new applications.

Identifying critical variables for software dependability has been explored from a software engineering perspective [23]. A critical variable, in this case, is based on its spatial and temporal impact, with respect to other software components. Similar to our work, their work also uses the static and dynamic properties of a program to quantify the impact of a variable. However, their technique also uses the failure rate of the variables in deciding if a variable is critical, which requires programmer knowledge and manual effort to calculate. In contrast, our technique is completely automated, and does not require any semantic information from the programmer.

Khudia et al. [19] use profile-based analysis along with symptom-detection to identify critical instructions for protecting against soft errors. They classify library and function calls as high-value instructions, and they tag as critical all instructions that produce the operands of the high value instructions i.e., those instructions in the backward slices of the high-value instructions. They perform memory and value profiling optimizations to reduce the overheads. However, they do not distinguish between EDC-causing and non-EDC causing errors, and hence their approach may perform wasteful detection and recovery for soft-computing applications.

Detector Placement Hari et al. [17] address the problem of detector placement (and detector derivation) for SDC-causing faults. The authors use a bottom up approach of analyzing the assembly code of specific programs to see what properties contribute to an SDC. Although we focus on EDCs, at a high-level, their work is similar to ours in terms of identifying program
properties that cause a specific failure type. However, their work differs from ours in three ways. First, though they investigate detector placement locations, their main focus is on reducing the performance overhead incurred by instruction replication. Second, they rely on program specific functions in four out of six benchmarks to develop customized detectors (e.g., bit reversal and exponential functions). It is not clear how representative are these functions of general programs. Third, because of the use of high coverage customized detectors, their approach requires fault injection and manual extraction of specific program characteristics, at the machine code level, which is expensive.

Pattabiraman et al. [32] develop a set of heuristics for strategic placement of detectors to detect crash causing errors with low detection latency. The heuristics are calculated using the dynamic dependency graph (DDG) of the program, from one or more input sets. While these metrics help in placing detectors to preemptively detect crashes, their coverage for SDC (and EDC) causing errors is low. Further, because their approach requires constructing the DDG apriori, it has high performance overheads.

Hiller et al. [18, 22] focus on error detector placement from a software engineering perspective. They compare various techniques for identifying detectors, under different error models. However, they require fault injections to guide detector placement, which can be time consuming, and further do not focus on EDC-causing errors.

Snap [5] automatically identifies critical regions in code by grouping related input bytes into fields. It relies on application code, and one or more inputs to see how targeted input fuzzing changes the behaviour of code.
1.5. Related Work

Code that causes large changes in output is classified as critical code, while code that induces small changes in output is classified as forgiving code. To some extent this is similar to our work on using program characteristics to identify detector placement points. However, their technique requires fuzzing, which is analogous to fault injection, and is hence time consuming.

Full duplication of programs using software redundancy will achieve close to 100% EDC coverage, at the cost of high performance overhead. However, there have been efforts to reduce the performance overhead using speculative redundant multithreading. An example is DAFT [45], in which the average performance overhead is reduced to 38%. DAFT overcomes this by speculatively checking the results between the original and duplicate threads, and reducing the average performance overhead to 38% by asynchronously checking the results. However, due to the shared memory model, DAFT does not replicate loads and stores, and may miss faults in those instructions. Also, as the focus is on detecting SDCs with high coverage, and there is no effort to differentiate EDCs from this set. Therefore, DAFT can incur high Non-EDC coverage.

**Effect of compiler optimizations** Prior work [13, 35] has focused on the effect of compiler optimizations on application vulnerability, i.e., the probability that a hardware fault affects the application and that fault leads to an application failure (see Equation 1.2). This is different from resilience, which is the probability of a failure, given that an error has occurred in the application (see Equation 1.1). Unlike vulnerability, resilience is a property of the application alone, and does not depend on hardware characteristics.

In addition to the focus on resilience, our work (see Chapter 7) differs
from the above ones in that it considers the effect of individual optimizations (e.g., Loop Invariant Code Motion) on the resilience of the application. On the other hand, the above studies look at the effects of collections of optimizations enabled by the optimization levels in gcc. This makes it difficult to pinpoint which optimization is responsible for the reduction in resilience (if any).

1.6 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 explains our fault injector used as part of our initial study and later for evaluating our technique. Chapter 3 explains our fault injection study. Chapter 4 presents our approach, while Chapter 5 introduces the experimental setup. Chapter 6 presents the evaluation, while Chapter 7 presents the effect of compiler optimizations on the error resilience of soft computing applications. Finally, Chapter 8 concludes and proposes some future directions.

1.7 Summary

In this chapter, we motivate our focus on hardware error detection through software solutions in a particular class of applications called soft computing applications. Hardware errors are increasing because of many factors such as reducing feature sizes and voltage scaling. Soft computing applications do not require precise outputs, and can tolerate most kinds of faults in code or data. We study the resilience of these applications to the faults that actually matter, i.e., the ones that cause Egregious Data Corruptions.
1.7. Summary

(EDCs). EDCs are unacceptable outcomes to the end user, that have large deviation in output. Our goal is to preemptively and selectively detect faults that lead to EDCs.

In the following chapters, we present our software based technique for identifying detector placement locations in the code for EDC causing faults. We formulate certain program level heuristics for identifying EDC prone regions of code, by performing fault injection using our compiler based fault injector LLFI.
Chapter 2

LLFI

In this chapter, we explain our fault injector tool - LLFI \[40\], that we built for our fault injection experiments as part of our initial study (Chapter \[3\]) and for evaluating our technique (Chapter \[5\]). We present the implementation of our fault injector using an example. We also compare the accuracy of our fault injector versus assembly level fault injection for soft computing applications.

2.1 Motivation

Fault injection experiments are performed at mainly four levels: source code, intermediate code, assembly and microarchitectural level. While fault injection at the source code level enables easy understanding of fault propagation, it is inaccurate in emulating hardware faults. This is because information available at lower layers such as load/store instructions and address computations, is unavailable at the source code level. Traditionally, fault injection is performed at the assembly code or microarchitectural level. Although these fault injections are more accurate than performing injections at higher levels, it is difficult to reason about fault propagation and their high level characteristics.
2.2 Methodology

We perform the fault injection at the LLVM intermediate code of applications. LLVM is a widely used compiler framework and has support for a variety of front end languages such as C and C++ [20]. LLFI works at the LLVM compiler’s intermediate code level, and allows fault-injections to be performed at specific program points, and into specific data types. It also enables tracing the propagation of the fault in the program by instrumenting the program at selected points. LLFI is closely integrated with the LLVM compiler, and can hence support a wide variety of programs.

We chose LLVM for our fault injection framework because:

1. Its intermediate code is a typed language, in which source-level constructs can be easily represented. In particular, it preserves the variable and function names, making source mapping feasible.

2. It has support for program analysis and transformations which makes it easier to study the effect of fault injection at a higher level than the assembly language, while still accounting for details not visible in the source code, such as address computation.

3. The lowering of the intermediate code to specific architectures is robust [41] - the analysis done at the IR level can be used at assembly level deterministically and with reasonable accuracy.

2.2 Methodology

LLFI injects a fault, i.e., a single bit flip into the destination register of exactly one dynamic instance of an instruction chosen at random (among
all the executed instructions), and classifies the outcome of the fault by comparing the final outcome with the fault free outcome. The fault-free or baseline outcome is obtained by running the original executable with the same input, but without any injected faults. The faulty outcomes are classified into Crash, Benign, EDCs and Non-EDCs. The EDCs are separated from the Non-EDCs based on the fidelity threshold value.

**Example:** We explain the fault injection process using the factorial program in Figure 2.1 as an example. The original IR for the corresponding C code is shown in Figure 2.2. The value of \( n \) (the number whose factorial is calculated) is assigned in line 7 of the C code. This corresponds to \( \%2 \) in line 6 of Figure 2.2. The basic block \( bb1 \) in lines 15-18 in the IR corresponds to the loop header at line 9 in the C code. The \( \text{phi}^{1} \) node at line 15 gets the value of the iterator variable \( i \), which is initialized to 1 (from basic block entry) or obtained from the incremented value within the loop at basic block \( bb \). Similarly, \( \text{fact} \) is also assigned the \( \text{phi} \) node value where it is either 1 (the initial value), or the value of \( \text{fact} \) from within the loop at line 10.

Figure 2.3 shows the original IR statically instrumented for fault injection. Each instruction with a return value, has a call to the fault injection function \( \text{injectFault} \). For example, lines 10 and 11 in Figure 2.2 have corresponding fault injection calls at lines 17 and 19 in the instrumented IR in Figure 2.3. The arguments of the \( \text{injectFault} \) function are the static fault ID, the type of call (fault injection call type has value 0), and the result of the instruction itself. This function flips a randomly chosen bit in

\(^1\text{Phi is a construct used in SSA form}\)
2.2. Methodology

Figure 2.1: C code for factorial program

```c
#include<stdio.h>
main(int argc, char *argv)
{
    int i, fact, n;
    n = atoi(argv[1]);
    fact = 1;
    for(i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    printf("%d\n",fact);
}
```

Figure 2.2: LLVM IR for the factorial program

```llvm
define i32 @main(i32 %argc, i8* %argv) nounwind {
  entry:
  %alloca point = bitcast i32 0 to i32
  %0 = getelementptr inbounds i8* %argc, i64 1
  %1 = load i8** %alloca, align 1
  %2 = call i32 (...) @printf(i8* %alloca) nounwind
  br label %bb1
bb1:
  %3 = mul nsw i32 %fact 0, %1 0
  %4 = add nsw i32 %fact, %1 1
  br label %bb1
bb1:
  %1.0 = phi i32 [ 1, %entry ], [ %4, %bb1 ]
  %fact.0 = phi i32 [ 1, %entry ], [ %3, %bb1 ]
  %2 = icmp slt i32 %fact 0, %2
  br i1 %2, label %bb1, label %bb2
bb2:
  %6 = call i32 (...) @printf(i8* %alloca) nounwind
  inbounds [4 x 10] i8* 0.str, i64 0, i64 0], i32 %fact 0) nounwind
  br label %return
return:
  ret i32 undef
}
```

Figure 2.3: LLVM IR of Figure 2.2 instrumented with fault injection calls

```llvm
define i32 @main(i32 %argc, i8* %argv) nounwind {
  entry:
  %alloca point = bitcast i32 0 to i32
  %0 = getelementptr inbounds i8* %argc, i64 1
  %1 = load i8** %alloca, align 1
  %2 = call i32 (...) @printf(i8* %alloca) nounwind
  inbounds [4 x 10] i8* 0.str, i64 0, i64 0], i32 %fact 0) nounwind
  %3 = call i32 (...) @printf(i8* %alloca) nounwind
  br label %return
return:
  ret i32 undef
}
```
the instruction’s result, and returns this faulty result. Next, all uses of the original instruction are replaced by the fault injected instruction. For example, line 15 in the original IR corresponds to line 23 in the instrumented IR, and %4 is replaced by the corresponding fault injection call %fi7. Since LLVM is a typed IR, we handle instructions of different types by having separate fault injection calls for each type.

At runtime, we first obtain the total number of dynamic instructions by running the original factorial IR. Second, for each fault injection run, we choose a random instruction instance from the set of all dynamic instructions for fault injection. This is done by the function initInjections inserted at the beginning of the program (at line 3 in Figure 2.3). Third, for each instruction, the injectFault function checks if the particular instance is the dynamic instance to be fault injected, and if so, flips a single bit in the instruction result. This fault-injected result is propagated to all the uses of this instruction in LLVM.

2.3 Accuracy of LLFI

While the LLVM intermediate code is close to the assembly code, it does not correspond one-to-one with the assembly language. We qualitatively assess the correspondence between the LLVM intermediate code and the assembly code for fault-injection purposes. The differences are presented in Table 2.1. We quantify the effect of these differences in Section 2.5.
### 2.3. Accuracy of LLFI

*Table 2.1:* Difference between LLVM intermediate code and Assembly Language, and the impact on fault injection

<table>
<thead>
<tr>
<th>LLVM Instruction</th>
<th>Assembly Language Instruction</th>
<th>Mapping (if possible)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The GetElementPtr (GEP) instruction does address computation which is supplied to the load and store instructions for memory access.</td>
<td>A set of add and multiply instructions that computes the address</td>
<td>A fault in the GEP instruction translates to a fault in one of the add or multiply instructions</td>
</tr>
<tr>
<td>The PHINode instruction is inserted when the stack allocation and deallocation is promoted to registers, to choose between values merging from different basic blocks.</td>
<td>Stores instructions in place of the PHINode instruction, and copy instructions in place of the source instructions in the corresponding basic blocks</td>
<td>A fault in the PHINode instruction translates to a fault in the store instructions, or the copy instructions corresponding to the sources of the PHINode instruction</td>
</tr>
</tbody>
</table>
2.3. Accuracy of LLFI

<table>
<thead>
<tr>
<th>Function call</th>
<th>PUSH/POP instructions for Caller/Callee saved registers before and after a function call, and Stack pointer stores return address</th>
<th>None since these instructions do not exist in the LLVM intermediate code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional branch instructions</td>
<td>Jump instructions where the Instruction Pointer register is fault injected</td>
<td>None since the branch instruction in LLFI is not fault injected, as it does not have a return value</td>
</tr>
</tbody>
</table>

**Comparison with flip-flop level fault injections:** Cho et al. [7] study the inaccuracies associated with fault injections at higher levels of abstraction, compared to performing fault injections at the flip-flop level. The latter is accurate and necessary for making claims about how effects of low level behaviours (e.g., cosmic ray flux) propagate to the higher levels of the system stack, and manifest as errors in the application.

However, we are studying the *relative effect of these hardware faults on the error resilience of soft computing applications* (see Equation 1.1 for resilience definition). We compare the relative resilience of applications protected by our detection technique in comparison to having no technique in
2.4 Implementation

In this section, we explain the implementation of our fault injector, and the assembly level fault injector we built for our accuracy analysis (see Section 2.5).

**LLFI:** We implemented the fault injector LLFI and the tracing mechanism as custom passes in the LLVM compiler version 2.9. First, the application source code is compiled into LLVM IR along with the `mem2reg` optimization (i.e., promote loads/stores to registers). Second, the IR is statically instrumented with calls to our custom fault injection function, as explained in Section 2.2. Third, the IR is statically instrumented with calls to the custom trace function, for control and pointer data (see Chapter 3 on details about the data types monitored).

**PIN Fault Injector:** We build a fault injector at the assembly code level using PIN [27], a binary instrumentation framework for X86 processors. We use the PIN fault injector to quantitatively validate the accuracy of LLFI versus assembly level fault injection. The PIN fault injector is based on the same fault model used for LLFI (see Section 3.1). We inject a single bit flip into the write register of one dynamic instance of an assembly instruction chosen at random from the set of all instructions at runtime. The number of fault injections and the fidelity threshold value were kept the same between
2.5. Results on LLFI Accuracy

In this section, we present our results on the accuracy of our fault injector tool LLFI compared to assembly level fault injection.

**Benchmarks:** We performed the fault injection on six applications of MediaBench I and II [15, 21]. Three of the benchmarks, namely JPEG, MPEG2 and H264 decoders, use Peak-Signal-to-Noise Ratio (PSNR) between the faulty and fault-free decoded images as the fidelity metric [25]. JPEG is an image decoder, while MPEG2 and H264 are video decoders. The other three benchmarks are speech decoders - G721, GSM and ADPCM, and they use Segmental SNR as the fidelity metric [25]. We use PSNR of 30, and Segmental SNR of 80 as fidelity threshold values to differentiate EDCs from Non-EDCs. We injected 1000 faults per benchmark using LLFI. The EDC rates for all benchmarks are within an error bar of 2.2%, at the 90% confidence level.

**Quantitative Validation of LLFI**

Figure 2.4 quantifies the difference in fault outcomes between fault injection done using PIN versus LLFI. For all benchmarks, the EDC and Non-EDC rates are similar or higher for LLFI versus the PIN injector. This shows the validity of performing fault injections at the intermediate code level. On average, LLFI has an EDC rate of 6.4% versus 3.3% for

---

2 We use Segmental SNR of 30 for ADPCM alone
2.5. Results on LLFI Accuracy

the PIN fault injector, and a Non-EDC rate of 42.8% versus 23.9% for the PIN fault injector. A significant fraction of faults in the PIN experiment are benign or result in crashes. The average EDC rate is skewed towards the high EDC rate in ADPCM (which is around 23% using LLFI). By excluding ADPCM, the average EDC rate is 3% for LLFI versus 2% for PIN. Unlike the other benchmarks, ADPCM has around 750 lines of code, with the main functionality of the benchmark concentrated in a single function. This function has many branch instructions, which leads to a high number of SDCs.

PIN Injector: To better understand the difference between the results of LLFI and PIN injector, we study the correlation between faults at specific registers and the fault outcomes in the PIN injector. Faults in the instruction pointer (IP) and the stack pointer (SP) registers predominantly resulted in crashes. However, a small number of faults in the IP register resulted in
2.6. Summary

EDCs (around 0.4% out of the total fault injections). These faults do not have a corresponding mapping in LLFI (difference 4 in Table 2.1).

The high number of benign outcomes in PIN was due to faults affecting the RFLAGS register. This register is written in the case of the test instructions which is the predicate used for conditional branching. RFLAGS has 64 bits, each representing specific flags or reserved bits. The flag usually tested is the zero flag in the corresponding conditional branch instruction. Hence, a single bit flip which affects only one of these flags, does not cause an error unless the zero flag bit is flipped.

2.6 Summary

In this chapter, we described the design and implementation of LLFI, our compiler based fault injector tool. LLFI performs fault injection at the LLVM compiler’s intermediate code level of the application. We validate the accuracy of LLFI compared to assembly level fault injection for soft computing applications and we find that LLFI is reasonably accurate. Using LLFI, we perform a preliminary study to understand the resilience characteristics of soft computing applications.
Chapter 3

Preliminary Study

This chapter describes our initial study for identifying potential locations to place detectors for EDC causing faults through fault injection experiments. We first present our fault model in Section 3.1 and describe the fault injection experiment in Section 3.2. We then describe the results of the fault injection experiment in Section 3.3. Based on these results, we develop heuristics for selectively finding the EDC causing data in the program, as explained in Chapter 4.

3.1 Fault Model

We consider transient hardware faults that occur in the processor. These are usually caused by cosmic ray or alpha particle strikes affecting flip flops and logic elements. These factors get exacerbated as the supply voltage is reduced for saving power in processors.

We consider faults that occur in the functional units, i.e., the ALU and the address computation for loads and stores. However, faults in the memory components such as caches are not considered, since these components are usually protected at the architectural level using ECC or parity. We do not consider faults in the control logic of the processor as this is a small portion
of the processor area, nor do we consider faults in the instructions, as these can be handled through control-flow checking techniques \cite{31}. As in prior work, we do not consider faults in floating point registers \cite{17} - this is part of future work.

3.2 Fault Injection Experiment

For the fault-injection experiments, we use LLFI, our program level fault-injection tool for performing the fault injection experiments (see Chapter 2). We performed the fault injection on six applications of MediaBench I and II \cite{15,21}. This is the same experiment explained in Section 2.5 with 1000 faults injected per benchmark.

After injecting the fault, we monitored the program at selected data items, in order to determine if the values of the data items exhibit a deviation from their fault-free values. In other words, we compare the value of the data items between the fault-free and fault-injected runs (we collect these values by instrumenting the program with the LLVM compiler). This information will help us determine whether to place detectors at the variable.

We split the results according to the type of the monitored data items, since we wanted to see if there was a correlation between the fault outcome and the type of data that exhibits a deviation from its fault-free value. Because we perform the injection and analysis at the LLVM intermediate-code level, we can track the detailed provenance of the data and its subsequent uses in the program. This also helps us formulate heuristics explained in Section 4.2.
3.2. Fault Injection Experiment

We chose the following data type categories for splitting the results in order to study the correlation with data types.

1. Pointer data: Prior work has found that there is a high probability of SDCs when the fault affects the lower order bits of the pointer variable [17].

2. Control data: Prior work has found that a fault in control data may cause a control deviation, which might egregiously affect the computation [36, 37].

We separated the faults injected into those that affected data items that were present in the backward slice of either of these two types of data. The backward slice of a particular variable consists of all instructions that affect the output of the variable through a control or data dependency [43]. A fault occurring in the backward slice of a variable would be likely to propagate to the variable, and hence placing a detector at that variable would likely detect the fault. This leads to four classification categories as shown in Table 3.1.
### 3.2. Fault Injection Experiment

*Table 3.1: Classification of faults according to the backward slices of data categories (explained using example in Figure 1.2)*

<table>
<thead>
<tr>
<th>Data Category</th>
<th>Explanation</th>
<th>Eg.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pointer and Control</strong></td>
<td>The fault (a) directly affects or propagates to a pointer, before affecting control data (deviation or backward slice) later in the execution OR (b) affects backward slice of control data (without causing a flip), and then propagates to pointer data</td>
<td>B1</td>
</tr>
<tr>
<td><strong>Pointer and No Control</strong></td>
<td>The fault affects pointer data, but is not present in the backward slice of control data</td>
<td>P1</td>
</tr>
<tr>
<td><strong>Control and No Pointer</strong></td>
<td>The fault either causes a branch flip without/before affecting any pointer data, or affects control backward slice without affecting any pointer data</td>
<td>B2</td>
</tr>
<tr>
<td><strong>Neither Control nor Pointer</strong></td>
<td>The fault (a) gets masked before being classified as belonging to the backward slice of control or pointer data OR (b) the failure outcome (crash) occurs before any deviation in the backward slice values of monitored data</td>
<td>-</td>
</tr>
</tbody>
</table>
3.3. Fault Injection Results

Figure 3.1 shows the average results across benchmarks for different failure types. From the figure, the average EDC rate across applications is 6.4%, while the Non-EDC rate is 42.8%. The crash rate is 22.5% and the benign fault rate is 28.3%.

From the overall results, one can observe that EDCs constitute a small, but non-trivial fraction of the application outcomes. In fact, only 8% of the errors that do not crash the application result in EDCs. This shows that blindly detecting all errors would result in significant wastage of energy and time, as many of them do not cause EDCs. This is the main reason why we develop targeted techniques for detecting EDCs in soft computing applications.
3.3. Fault Injection Results

We observe the following trends about the relationship between the data items monitored and the fault outcome from Figure 3.1. We use these observations for formulating heuristics in Section 4.2.

**R1. Control Non Pointer** Faults in data items belonging to the control backward slice (third category in Table 3.1) are highly likely to lead to EDCs. These data items are usually loop termination conditions, and are further discussed in Section 4.2.

**R2. Pointer Non Control** Data items that are in the backward slice of pointer data, but not control data (i.e., pointer non-control) are responsible for most crashes. However, such faults can also result in SDCs (i.e., both EDCs and non-EDCs), especially if the fault affects the low-order bits of the pointer variable.

**R3. Non Control Non Pointer** Faults in data items that do not belong to the backward slice of control or pointer data are mostly benign. Therefore, we do not consider this data category in formulating heuristics.

**R4. Control Pointer** Although only a small fraction of faults cause deviations in the backward slices of both control and pointer data (around 5.8%), there is a high probability of such faults resulting in an SDC (e.g., around 62%). We take this into account when formulating heuristics in Section 4.2.
3.4 Summary

In this chapter, we study the error resilience of soft computing applications. We perform fault injection experiments using LLFI, and identify correlations between certain data types (control and pointer data) and fault outcomes. Based on the results of our fault injections, we formulate program level heuristics in the following chapter, to identify EDC prone regions of code.
Chapter 4

Approach

In this chapter, we first list the assumptions we make in our work, and then describe our approach for identifying detector placement locations for pre-emptively and selectively detecting EDC causing faults. We develop heuristics for identifying these locations, and incorporate these heuristics in our automated detector placement algorithm.

4.1 Assumptions about Soft Computation

We make certain assumptions about soft computing applications in our work.

Relaxed Correctness: These applications have a relaxed notion of correctness that is defined by the fidelity threshold value. However, we do not relax it to the point that only outputs close to crashes are classified as EDCs. In such a case, it is better to have no detection technique in place. This is because the percentage of such EDCs would be much lower and any detection technique including prior work would lead to wasteful detection. Even with our notion of relaxed correctness, the percentage of EDC outcomes is only around 6% (see Chapter 3). In our evaluation, we start with a relaxed notion of correctness and perform a sensitivity analysis under varying fidelity threshold values, i.e., by imposing stricter constraints
4.2. Heuristics

We formulated heuristics to identify detector placement points for EDCs, on the basis of the fault injection experiments in Chapter 3. Our heuristics are generic, in that they can be applied to any soft-computing application, and do not require semantic knowledge of the application. Further, while we use fault injection to formulate the heuristics, the heuristics themselves do not need fault injection to be applied to new applications.

All these heuristics have the common characteristic of being dependent on the size of the data being affected, either within the branches or in downstream computations. We unify these heuristics using a ranking expression in our algorithm explained in Section 4.3.

We explain the heuristics with the code in Figure 4.1 as a running example. This code is based on the MPEG video decoding benchmark from the Mediabench benchmark suite. However, for elucidation purposes, we
4.2. Heuristics

have added extra code to these functions (we explain what these are later). The `store_ppm_tga` function stores the decoded image in a `ppm` file. The `Show_Bits(N)` function returns the next N bits of the image, without advancing the pointer. The `conv422to444` function is the same function described in Section 1.4.

```
void conv422to444(char *src, char *dst, int width, int height, int offset){
    ...
    w = width>>1;
    if(dst < src + offset) //B1
        return;
    ...
    for(j=0; j < height; j++) { //B2
        for(i=0; i < width; i++) { //B3
            i2 = i<<1;
            im1 = (i < 1) ? 0 : i-1; //B4
            ...
            dst[i2] = Clip[(21*src[im1])>>8]; //P1
        }
        if(j + 1 < height) { //B5
            src += w; //P2
            dst += width;
        }
    }
    ...
}

void store_ppm_tga(int width, int height){
    int i, j, singlecode;
    char *u444[NUMFRAMES];
    int *code[NUMFRAMES], codeframes[NUMFRAMES];
    ...
    //int *bitlocn[NUMFRAMES] is global
    for(i=0; i < NUMFRAMES; i++) { //B6
```
4.2. Heuristics

We divide the problem of formulating heuristics for identifying detector placement points into two steps. First, we identify functions in the program that are likely to result in EDCs when affected by faults. Second, we identify statements (and variables) within a function at which detectors need to be
4.2. Heuristics

placed in order to detect EDCs.

4.2.1 Step 1: Function Identification

We first identify program functions in which we need to place detectors, based on whether the functions have side effects. A side-effect free function has the following two characteristics, both of which must be satisfied:

1. Statements within these functions do not modify global variables, files and pointers, though they may read them.

2. The functions have a return value and this is the only result of the function used by its caller function.

We call such functions Optimized EDC Functions (OEF). For example in Figure 4.1, Show_Bits() is an OEF, as it satisfies the conditions outlined above. Once an OEF call is identified as EDC-causing, it suffices to place a detector at the return value of the particular call. No other detectors are required for the OEF, and hence the name Optimized EDC. We find that EDCs are caused by only certain calls to OEFs, and we formulate a heuristic for identifying such OEF calls.

H1: The likelihood of an EDC due to a fault in an OEF increases as the amount of data affected by its return value increases.

We formulated this heuristic based on the results R1 and R2 (see Section 3.3). By the definition of OEFs, the data modified within these functions is local to the function. Therefore, the data modified by an OEF call is dependent on the propagation of the function’s return value. For example, Show_Bits(), which is an OEF, is called at three places in the code, namely
4.2. Heuristics

C0, C1 and C2\(^3\) When a fault occurs in the OEF, the return value of the function call at C1 affects only one element of the 2D code array. This fault does not cause an EDC. On the other hand, the function call in C0 is a loop carried dependency, and the singlecode variable is assigned to the elements of array codeframes in the outer loop. The return value from the C0 call thus influences a larger amount of data than the return value from call C1. Therefore, we will place a detector on the return value in call C0, but not on the return value in call C1.

Based on the heuristic H1 above, we identify the calls to OEFs on whose return values we place detectors. Note that this heuristic only applies to OEFs called within loops. When the OEF is not called within a loop, we do not place any detector at the return value. This is because such faults usually cause an EDC when they propagate to branches, and would be caught by detectors at those branches. For example, neither the call C2, nor its caller function store.ppm.tga is called within a loop. Hence, we do not place a detector after call C2.

The remaining functions are side-effect causing functions. For example, these are conv422to444 and store.ppm.tga, since they do not satisfy the conditions required for being classified as an OEF. We elaborate the heuristics applicable to such functions in the following section.

\(^3\)This function is called only in the manner specified in C2, in MPEG. We add the remaining calls, to explain examples seen across other applications.
4.2. Heuristics

4.2.2 Step 2: Data Categorization

Within functions that are not OEFs, we found that faults affecting certain control and pointer data are highly likely to cause EDCs (based on results R1, R2 and R4).

Control Data: We formulated the heuristics for control data by analyzing the results R1 and R4. Control data can be divided into loop or function terminating branch conditions, and other branches, i.e., those that do not terminate loops or functions. For example, B1 is a function terminating branch, while j < height at B2 is a loop terminating branch. The heuristics are based on faults that either directly affect or propagate to these branches, and cause the branch to flip.

H2. The EDC causing nature of the loop terminating conditions decreases, as we go deeper within nested loops. We formulated this heuristic based on the result R1. This is because the amount of data modified by outer loops is much larger than the data modified by inner loops. For example, a fault at branch condition i < NUMFRAMES at B7 has a higher likelihood of causing an EDC than one at branch condition j < width at B8, as it affects more elements of the array code.

H3. The likelihood of an EDC due to faults at function terminating conditions decreases as the inter-procedural loop nesting level increases, and as the amount of data affected in downstream computations within that function decreases.

We formulated this heuristic based on the result R1. The EDC causing

---

4In our preliminary study, we found that faults causing branch flips are much more predominant than those that do not cause a flip for control data.
4.2. Heuristics

nature of function terminating branches decreases, as the inter-procedural loop level increases. For example, the function terminating branch B1 has a loop level of 1, since the function conv422to444 is called within a loop at C3. Also, a fault at B1, causing a branch flip to \textit{true}, abnormally terminates function execution, thereby missing the loop computations at B2. These two factors, i.e., the downstream loop computations and the low loop nesting level, contribute to a high likelihood of an EDC, when the fault occurs or propagates to B1. In cases where loops do not follow the function terminating conditions, we see that the faults do not cause EDCs.

\textbf{H4.} \textit{Branch conditions that do not terminate functions or loops are likely to cause EDCs if and only if the amount of data affected within the body of the branch is large.}

We formulated this heuristic based on results R1 and R4:

1. When the branch body consists of assignments to pointers, or several elements of an array or aggregate structure, a fault occurring at the branch results in an EDC. In the above example, we place a detector after branch B5 since the body of the branch changes the pointers \texttt{src} and \texttt{dst}.

2. When the branch body consists only of a change to a single element of an array, or some local variable, a fault in the branch results in an SDC, but not an EDC. For example, the ternary condition \texttt{i < 1} at B4 is a Non-EDC causing branch, since it only changes the index of one element in the array \texttt{src}, thereby corrupting the value of one element of array \texttt{dst}. 

41
3. When a branch condition (that does not terminate loops or functions) has loops within it, a fault at the branch condition has a high likelihood of causing an EDC. This is because the amount of data modified is large in the loop body. For example, a fault causing a branch flip at B9 has a high likelihood of causing an EDC, since it causes the loop at B10 to be skipped, thereby affecting the computation of the entire array u444.

**Pointer Data:** Examples are pointer dereferences, accesses to specific elements within aggregate structures, and pointer assignments or arithmetic. Result [R2] shows that the number of faults leading to crashes, SDCs and EDCs are high for pointer data that do not cause any control deviation. This pointer data usually occurs within loop bodies. As prior work finds [17], crashes are caused when a bit flip occurs in the high order bits of the memory access, whereas SDCs are caused when the bit flip is in the low order bits. However, we find that some pointer address computations are more likely to cause an EDC, and we formulate a heuristic to identify these computations.

**H5. Faults in the low order bits of pointers pointing to larger sized data have higher likelihood of causing an EDC.** We formulated this heuristic based on result [R2]. For example, faults in the low order bits of pointer data for src at P2 causes an EDC. However, a fault at the lower bits of the Clip, src or dst array indices at P1 causes an SDC, but not an EDC.
4.3 Approach

In this section, we first present the usage model for our technique, and then discuss our algorithms to identify program locations for high coverage detectors for EDC causing errors. These are based on the heuristics we developed in Section 4.2. Note that for any new soft computing applications, we do not perform fault injections or develop new heuristics. We directly apply our automated algorithm to these applications.

Usage Model: The goal of our technique is to preemptively detect EDC causing faults in soft computing applications, under a given performance overhead that the user is willing to tolerate. The technique requires as inputs from the user: (a) the application source code, (b) the maximum permissible performance overhead, and (c) the application’s execution profile, under representative inputs.

The workflow of our technique is outlined in Figure 4.2 and consists of three steps. First, we compile the application source code using a standard compiler into an Intermediate Representation (IR). The IR should retain type information from the source code, and should be in Static Single Assignment (SSA) form [10]. SSA requires a variable be assigned exactly once.
in the program i.e., every variable in the program has a unique instruction that assigns to it. Second, we rank the application’s data according to their likelihood of causing an EDC using an EDC ranking algorithm. Third, we choose the optimal data set for detector placement under the given performance overhead bound, using a selection algorithm that combines the obtained EDC ranks and the runtime profiling information. We describe the second and third phases of Figure 4.2 in Sections 4.3.1 and 4.3.2.

4.3.1 EDC Ranking Algorithm

In this phase (step 2 in Figure 4.2), we first identify the initial dataset, i.e., the list of potential EDC causing data items based on the heuristics in Section 4.2. We then extract certain common attributes of these data items using static analysis. Finally, we formulate a ranking expression using these attributes, and rank these data items using our ranking expression.

**Initial DataSet:** The initial dataset consists of all the data categories identified through heuristics H1 to H5. These are OEF calls, control data and pointer data. Note that this dataset contains EDC as well as Non-EDC causing data items. Using the heuristics, we formulated a ranking metric to rank these data items based on their tendency to cause EDCs (when faulty).

**EDC Rank Characteristics:** We discuss the characteristics of the EDC rank, and the rationale behind it. In Section 4.2, we found that data items that affect a larger amount of data have a higher likelihood of causing EDCs. Hence, the EDC rank should be higher for data items affecting larger-sized data. In other words, for any given data item d, the branch b it is control-dependent on, has a higher EDC rank than d. For example in Fig-
4.3. Approach

Table 4.1: Attribute Values and Static EDC Rank for data items from example in Figure 4.1. Static EDC rank is calculated using equation [4.1] with the values of $\alpha = 4$, $\beta = 3$, $\mu = 2$, and $\gamma = 1$. Higher EDC rank implies higher likelihood of EDC.

<table>
<thead>
<tr>
<th>Data Item</th>
<th>OuterLoop Level</th>
<th>InnerLoop Level</th>
<th>DomLoop Level</th>
<th>Data-Within Rank</th>
<th>EDC Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>B2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1667</td>
</tr>
<tr>
<td>B5</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>B6</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>0.1667</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1/3</td>
<td>0.1667</td>
</tr>
<tr>
<td>B9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

We ensure these characteristics are satisfied through the computation of the attributes in the EDC rank equation, as explained below.

**Attribute Extraction:** The EDC rank of a data item depends on various attributes, which are extracted through static analysis of the program. The attributes and their values for the example in Figure 4.1 are shown in Table 4.1. We explain the attributes below, using the example.

1. **OuterLoop Level** - The maximum level of loop nesting this particular data item is nested at. We extract this attribute based on heuristic H2. The outermost loop is at level 1, the next loop at level 2, and so
4.3. Approach

on. For data items that are not loop or function terminating, the loop level is one level more than the number of loops it is nested within. This is to satisfy the EDC rank characteristic, and unify the attribute extraction across all data items. For example, branch B4 in Table 4.1 has outerloop level of 3.

2. **InnerLoop Level** - The maximum level of loop nesting within this data item. We extract this attribute based on heuristics H2 and H4. For example, branches B2, B6 and B9 have an innerloop level of 1.

3. **DomLoop Level** - The maximum level of loop nesting dominated by this particular data item, but excluding the innerloop level. The data item $d$ dominates a loop if every path in the control flow graph from the start node to the loop should pass $d$. We extract this attribute based on heuristic H3. For example, the value for the function terminating condition B1 is shown in Table 4.1.

4. **DataWithin** - The amount of data affected by the data item. This applies to OEF calls, branches that are not loop or function terminating, and pointer data. We extract this attribute based on heuristics H1, H4 and H5. For pointer assignments and arithmetic, the numerical value of datasize is equal to the level of pointer indirection. In case of array accesses, the datasize is computed as $1/(1 + \text{number of array indices})$. For example, the datasize for both pointer data P2, and OEF calls C0 and C1 is shown in Table 4.1. Note that fractional values for

---

5 The attributes InnerLoop and DomLoop Level, already estimates the data affected by terminating conditions.
4.3. Approach

the datasize might be masked in some cases due to the minimum value being 1 in the numerator. However, we find that few such cases affect the final value of the EDC rank in practice.

Note that the first three attributes refer to the maximum nested level of the loops, and not the number of loops. For scalability reasons, we only consider the intra-procedural loop nesting level when computing the outerloop level attribute for the EDC data set. This does not affect our results for most benchmarks.

**Static EDC Rank Expression:** The EDC rank of a data item is the likelihood of an EDC outcome, given that a fault occurs at the data item or propagates to it. We formulate the rank expression using the attributes identified before:

$$\frac{\max(\alpha \ast InnerLoop + \beta \ast DomLoop + \gamma \ast DataWithin, 1)}{\max(\mu \ast OuterLoop, 1)}$$

(4.1)

where $\alpha$, $\beta$, $\gamma$ and $\mu$ are parameters quantifying the importance of the respective attributes, i.e., InnerLoop level, DomLoop level, DataWithin and OuterLoop level. To avoid zero values in the numerator and denominator, we assign the minimum value to be 1 in both parts.

We followed an educated trial and error method to assign the values for $\alpha$, $\beta$, $\gamma$ and $\mu$. The values assigned are $\alpha = 4$, $\beta = 3$, $\gamma = 1$ and $\mu = 2$.

We explain the assignment of these values here. Recall that the EDC rank is higher for data items affecting larger sized data. The impact on the amount of data affected, is much higher for the first three attributes ($OuterLoop$, $InnerLoop$ and $DomLoop$) than for the $DataWithin$ attribute.
4.3. Approach

This is because we speculate much higher amount of data is affected within loops than outside loops. Hence, we assign the lowest value for $\gamma$, which is set to 1. Further, as we go deeper down nested loops, the likelihood of an EDC outcome due to a fault in the data item decreases. Hence, we assign the highest value for $\alpha$, the coefficient of the $InnerLoop$ attribute, which is 4. The $DomLoop$ attribute has the next highest value followed by $outerloop$. So, we assign the values $\beta = 3$, and $\mu = 2$. We have experimented with other assignments in relation to each other, and find that the above assignment is the optimal one.

4.3.2 Selection Algorithm

In this phase, i.e., step 3 in Figure 4.2, we identify the optimal set of locations to place detectors in the program based on the EDC rank (from the previous phase), the allowed performance overhead and execution profile of the application. We use the profile data to maintain the bound on the performance overhead specified by the user, while accounting for the likelihood of a fault affecting the data item. We obtain the profile data by running the application with representative inputs provided by the user (see Chapter 5).

We model the problem of selecting the EDC data items as the 0-1 knapsack problem [9]. Each EDC data item $d$ has an associated weighted EDC rank $d_{wrank}$ (the objective function we maximize) and a performance overhead $d_{po}$, measured as the number of extra instructions that would need to be executed if the element is selected. Our goal is to select the items to put into the knapsack to maximize the rank subject to a given performance overhead. The weighted EDC rank is calculated using the following
4.3. Approach

equation:
\[ d_{\text{wrank}} = \left( \text{norm}(d_{\text{edcrank}}) + 1 \right)/F_{\text{funcrank}} \quad (4.2) \]

where \( F \) is the function containing the data item \( d \). The normalization function \( \text{norm} \), converts the \( \text{edcrank} \) (obtained from previous phase) to a value between 0 and 1. The \( \text{funcrank} \) is the rank of the function in descending order of their execution time. We choose the set of detector locations (the knapsack), using the following criteria:

\[
\text{maximize}(\Sigma d_{\text{wrank}}) \text{ such that } \Sigma (d_{\text{po}}) \leq P \quad (4.3)
\]

where \( P \) is the user specified maximum performance overhead.

A naive approach to solving the knapsack problem is a greedy one of choosing the item with the maximum weighted rank that satisfies the performance overhead constraint. However, a naive greedy algorithm may make a sub-optimal decision in choosing data items as it does not have a lookahead capability. We use a variant of the greedy algorithm that has a parameter controlling the function rank, and a lookahead window to avoid making a short-sighted, sub-optimal decision.

We explain the algorithm using an example. Let us consider five functions A, B, C, D and E, whose execution times are 10, 8, 6, 4 and 1 milliseconds, respectively\textsuperscript{6}. If we used a naive greedy algorithm, then the \( \text{funcrank} \) would be simply incremented as function execution times decreased. In this case, A, B, C, D and E would have respective \( \text{funcranks} \) of 1, 2,
4.3. Approach

3, 4 and 5. The selection algorithm would start filling the knapsack with data items of A in descending order of \textit{edcrank}, followed by that of B, and so on, until the maximum performance overhead \( P \) is reached. Hence, the Non-EDC data in function A will get included, and we may miss the EDC data in the remaining functions, leading to a sub-optimal solution.

To overcome this problem, we use a \textit{funcrange} parameter to increment the function rank. All functions having execution times within the \textit{funcrange} have the same function rank. We also use a lookahead window with functions having the next higher function rank. We explain how these ranks are obtained in our algorithm in Figure[4.3]. Assuming \textit{funcrange} with value 2, then functions A, B and C have a function rank of 1, D has a rank of 2, and E has a rank of 3. The selection window has functions A, B and C, while the lookahead window contains function D. Now, the knapsack is filled in descending order of \( d_{wrank} \) (where \( d \) is data items of \( A, B, C \) and \( D \)) until all the data items in the selection window are added. Next, the selection window slides ahead to D, and the lookahead window slides to E. The same process of filling the knapsack and sliding the window is repeated, until \( P \) is reached. As the \textit{funcrange} parameter increases, more functions would have the same function rank. Hence, the choice of detector locations would be based on a larger set of data, and hence be more optimal than a naive greedy algorithm.

```plaintext
float funcrank = 1;
int funcrange = N;
map funcrankmap;
map EDCrankmap;
int main()
```
4.3. Approach

```plaintext
map weightedrankmap;
function topFunc = Function with max exec time;
for each function 'F' ranked in decreasing order of execution times{
    if (F is an OEF)
        continue;
    if (topFunc.execTime/F.execTime > funcrange) {
        funcrank++;
        topFunc = F;
    }
    funcrankmap[F] = funcrank;
}
for each dataitem 'd' in initial DataSet{
    float weightedrank = calculateweightedrank(d);
    weightedrankmap[d] = weightedrank;
}
```

```plaintext
float calculateweightedrank(dataitem d) {
    Function F = d.getFunction();
    return (\( \frac{\|\text{EDCrankmap}[d]\| + 1)}{\text{funcrankmap}[F]} \));
}
```

Figure 4.3: Pseudo-code to show the calculation of weighted rank using `funcrange` of `N` where `EDCrankmap` is map of static EDC ranks for all data items using equation 4.1

The algorithm to calculate the weighted EDC rank using `funcrange` of `N` is presented in Figure 4.3. It considers the functions in the program in decreasing order of their execution times. All functions within the `funcrange` have the same function rank. When a function whose execution time is outside the parameter is encountered, the function rank is incremented. If a function is an OEF, it is skipped (see Section 4.3.1). After calculating the
weighted EDC ranks for all the data items, the final set of EDC detector locations is computed using equation [4.3]

### 4.4 Summary

In this chapter, we present our technique for preemptively and selectively detecting EDC causing faults. We formulate five program level heuristics that are unified based on a ranking metric and incorporate them in our automated detector placement algorithm. The algorithm is based on static analysis of the code and runtime profiling. It identifies the set of detector placement locations under a tolerable performance overhead bound specified by the user. In the following chapter, we present our experimental setup for evaluating our technique.
Chapter 5

Experimental Setup

In this chapter, we present the implementation details of our technique, followed by the benchmarks, the fidelity thresholds and the evaluation metrics.

5.1 Implementation

We implemented the EDC ranking and the selection algorithm (steps 2 and 3 in Figure 4.2) as custom passes in the LLVM compiler version 2.9. First, the application source code is compiled into LLVM Intermediate Representation (IR) along with the mem2reg optimization (i.e., promote loads/stores to registers). Second, the IR is statically (a) analyzed to compute the static EDC rank for the EDC dataset, and (b) instrumented to place detectors identified using profile data under the given performance overhead bound. Third, the instrumented IR is compiled into machine code using the LLVM compiler.

For the profile data, we need the user to provide representative inputs. However, the inputs are only used for calculating the performance overhead and the function rank. We have verified that the variation in EDC coverage

\[\text{We wrote a custom pass for obtaining profile data and for measuring the performance overhead, using LLVM basic block profiling pass.}\]
is minimal across the provided inputs for the benchmarks we studied. We used `funcrange` of 5 in our experiments based on coverage results obtained by varying its value (see Section 6.3). The time required for our custom passes, is on average less than three seconds across the benchmarks.

The error detectors are derived by replicating the static inter-procedural backward slice of the EDC data item, and placing a comparison statement after the copy of the item. This is similar to the ideal detectors used in prior work [18, 32, 36]. We do not consider reaching stores (for loads), and function pointers when computing the backward slice. Instead, we simulate these detectors by instrumenting the IR with trace calls at the locations chosen for detector placement. Hence, the coverage may be lower with actual detectors based on the backward slice. These trace calls record the values of the EDC data in a file, and a fault is detected if the fault-free and faulty trace files differ. The fault-free trace file is obtained by running the instrumented program on the same input, with no faults injected.

We measure the performance overhead of our detectors as the dynamic execution overhead of the extra code added, i.e., the replicated code and comparison statements. Equation 5.1 is the total performance overhead of our technique with \( n \) detectors, each having a static backward slice of size \( m_i \). The dynamic count of the instructions is obtained through the execution profile of the application.

\[
P.O. = \sum_{i=1}^{n} \sum_{j=1}^{m_i} \text{dynamic count of instruction } j
\]  

(5.1)

We assume that faults do not affect detectors, and hence we do not inject
faults into them. This is because we assume that only one fault occurs in each run of the application and a fault in the detector does not affect EDC coverage, as the worst outcome of such a fault is that it stops the program, and does not cause an EDC.

5.2 Benchmarks

We use four applications from Parsec [3], and two from Mediabench [21] for evaluating our technique. These are a mix of financial, multimedia and VLSI CAD applications, and have been used as soft computing applications in prior work [8, 25, 28, 36]. The benchmark characteristics are explained in Table 5.1.

The majority of the programs are different from what we chose in our initial study, in which we only use Mediabench. We use only two programs from Mediabench (MPEG and JPEG) out of the six from our initial study. We do not use G721 and GSM, because their fidelity metric values show very slight variation, making it difficult to separate the EDCs from Non-EDCs, even manually. ADPCM is a small benchmark program with 740 lines of code, while H264Dec overlaps significantly with the Parsec benchmark X264, and hence ADPCM and H264Dec are skipped.

The other four programs are from the Parsec suite. Canneal is from Parsec 3.0, while the remaining three are from Parsec 2.1. We made small changes to some of the benchmark programs as follows: (1) For Blacksc-holes, we removed a loop that artificially increased the execution time and served no other purpose (also found by prior work [28]), (2) For Canneal,
5.2. Benchmarks

we applied a patch to reduce the load times on the serial version [34], (3) For Swaptions, we modified the code to include the first 240 trials of the monte-carlo simulation. We also removed specific assertion checks on function return values, as we do not need to ’check these checkers’. We did not find such code in any of the other benchmarks.

Table 5.1: Characteristics of Benchmark Programs. Higher distortion (scaled difference) is more egregious, lower PSNR is more egregious.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input</th>
<th>Fidelity Metric (threshold value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlackScholes</td>
<td>Compute option pricing using Black-Scholes Partial</td>
<td>Sim-large</td>
<td>Scaled difference of option prices (0.3)</td>
</tr>
<tr>
<td>(1661)</td>
<td>Differential Equation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X264</td>
<td>Media Application performing H.264 encoding of video</td>
<td>test</td>
<td>Mean distortion of PSNR (as measured by H.264 reference decoder) and the encoded video’s bitrate (0.017)</td>
</tr>
<tr>
<td>(37454)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 5.2. Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Fidelity Metric</th>
<th>Metric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canneal</td>
<td>Simulated cache-aware annealing to optimize routing cost of a chip design</td>
<td>Sim-dev</td>
<td>Scaled difference of routing cost between faulty and original version (0.026)</td>
</tr>
<tr>
<td>(4506)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swaptions</td>
<td>Price portfolio of swaptions using Monte Carlo Simulations</td>
<td>Sim-small</td>
<td>Scaled difference of swaption prices (0.00001)</td>
</tr>
<tr>
<td>(1428)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG</td>
<td>Image Decoder</td>
<td>testing.jpg</td>
<td>PSNR between faulty and fault-free decoded images (30)</td>
</tr>
<tr>
<td>(30579)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPEG2</td>
<td>Video Decoder</td>
<td>mei16v2.m2v</td>
<td>PSNR between faulty and fault-free decoded image set (30)</td>
</tr>
<tr>
<td>(9832)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fidelity Metrics and Threshold Values:** We use the QoS metrics from prior work [28] as the fidelity metrics for the Parsec benchmarks. We distinguish EDCs from Non-EDCs using the fidelity threshold value (mentioned in parantheses in column 4 of Table 5.1). This threshold value does not change between inputs. The distortion or scaled difference is the difference in absolute values between faulty and original fault-free value divided
5.3 Coverage Evaluation

by the original fault-free value. For the Parsec benchmarks, we chose the fidelity threshold value such that 30% of the most egregious deviations from the SDC set are classified as EDCs. For MPEG and JPEG, we performed manual inspection of all the faulty outputs, and we noticed that EDCs were caused when the PSNR value was below 30, i.e., the images were severely distorted. Hence, we choose the value 30 as the fidelity threshold for these two programs.

5.3 Coverage Evaluation

We evaluate our technique by performing fault injection on the benchmark programs in Table 5.1. We use our LLVM compiler based fault injector LLFI (explained in Chapter 2) to perform the injections. We classify the outcomes as Crash, Benign, EDC and Non-EDC. The applications are run using the LLVM Just-In-Time (JIT) compiler with the default optimization level of O2. We injected 2000 faults per benchmark. The EDC rates are statistically significant within an error bar of 1.32% at the 95% confidence level.

We inject only one fault in each run, as we assume that transient faults are relatively rare events compared to the total execution time of an application. All injected faults are executed, i.e., the instruction into which the fault was injected is executed by the program.

We measure the coverage under varying bounds on performance overheads, i.e., 10%, 20% and 25% (provided by the user)\(^8\) The EDC coverage

---

\(^8\)We also measured coverage under 15% performance overhead, but do not present the results as they follow the trend of increasing coverages with higher performance overheads.
is the fraction of detected EDCs out of total EDCs, while the Non-EDC and benign coverage is the fraction of detected Non-EDCs and Benign faults out of the total Non-EDC and Benign faults. We do not consider crashes as they are easily detected by program termination.

5.4 Summary

In this chapter, we discuss the implementation of our technique, with the algorithms implemented as custom passes in the LLVM compiler. We present the six benchmarks used in our evaluation, along with their fidelity metrics and threshold values for separating EDCs from Non-EDCs. Majority of the benchmarks are different from those used in our preliminary study. We measure the preemptive property of our technique using the EDC coverage metric, and the selective property using the Non-EDC and benign coverage. Higher the EDC coverage, and lower the Non-EDC and benign coverage, the better our technique. We use these metrics for evaluating our technique in the following chapter.
Chapter 6

Results

In this chapter, we present the error outcome rates for the six benchmarks, followed by the coverage for EDC, and Non-EDC and Benign faults under varying performance overheads. We then study the effect of varying the funcrange parameter on the EDC coverage, and present the EDC coverage under varying fidelity threshold values. We also present a quantitative comparison between our technique and a technique proposed in prior work \[36\]. Finally, we discuss the advantages and the threats to validity of our technique.

6.1 Error Outcome Rates

Table 6.1 shows the Crash, Benign, EDC, and Non-EDC rates for the fault injection experiments across the six programs. The average EDC rate across these applications is 4.03\%, while the average Non-EDC and Benign fault rate is 57.57\%. Although, this may seem to suggest that EDCs are not very important, one should keep in mind that these constitute the worst outcomes of the application. Further, the average Non-EDC rate is 21\%, which is five times as much as the EDC rate. Hence, existing techniques that detect SDCs with high coverage will not be efficient for soft computing.
6.2 Coverage Under Varying Performance Overheads

applications, because these techniques would also detect Non-EDCs with high coverage resulting in wasteful detection and recovery (we compare our technique with one such technique in Section 6.5).

Table 6.1: Percentage of Error outcomes in each benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Crash (%)</th>
<th>Benign (%)</th>
<th>EDC (%)</th>
<th>Non-EDC (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlackScholes</td>
<td>51.52</td>
<td>13.25</td>
<td>10</td>
<td>25.23</td>
</tr>
<tr>
<td>X264</td>
<td>28.4</td>
<td>64.9</td>
<td>2.72</td>
<td>4.53</td>
</tr>
<tr>
<td>Canneal</td>
<td>53.25</td>
<td>37.87</td>
<td>2.9</td>
<td>5.98</td>
</tr>
<tr>
<td>Swaptions</td>
<td>42.05</td>
<td>48.46</td>
<td>2.57</td>
<td>6.92</td>
</tr>
<tr>
<td>JPEG</td>
<td>29.27</td>
<td>30.38</td>
<td>4.03</td>
<td>36.27</td>
</tr>
<tr>
<td>MPEG2</td>
<td>25.85</td>
<td>22.83</td>
<td>2.01</td>
<td>49.37</td>
</tr>
<tr>
<td>Average</td>
<td>38.39</td>
<td>36.19</td>
<td>4.03</td>
<td>21.38</td>
</tr>
</tbody>
</table>

6.2 Coverage Under Varying Performance Overheads

EDC Coverage: Figure 6.1 shows the absolute EDC coverage across programs for different overheads. The average EDC coverage across the benchmarks is 82% at 10% overhead, 85% at 20% overhead, and 86% at 25% performance overhead. All applications except for Swaptions, have an EDC coverage of 80-100% (average being 96%) at 25% overhead. Hence, our technique detects EDC causing faults with high coverage (above 80%) in five out of six applications, with low overheads (10%).
6.2. Coverage Under Varying Performance Overheads

The lowest EDC coverage of our technique is for the Swaptions program (45%). It is interesting to note that Swaptions has a relatively low EDC rate of 2.5%. On further investigation, we found that many EDCs are caused by faults in the uniform random number generator function \texttt{RanUnif()}. The values returned by \texttt{RanUnif} are used in the rest of Swaptions as an input for Monte-Carlo simulations. However, this location is not chosen by our detector placement algorithm under the given performance overhead bounds. Our technique protects this function call only at 35% performance overhead bound, at which point the coverage increases to 80%. We believe this is an anomalous case as we do not see this behaviour in any of the other five benchmarks.

**Non-EDC and Benign Coverage:** Figure 6.2 shows the Non-EDC and Benign coverage using our technique. Lower coverage is better as benign and Non-EDC faults are tolerated by the user, and we perform wasteful
6.2. Coverage Under Varying Performance Overheads

recomputation if these faults are detected as EDC causing faults and recovered from. The average coverage is 10%, 16% and 17.6% under respective performance overheads of 10%, 20% and 25%. Further, the average benign fault coverage is lower than the Non-EDC coverage.

![Figure 6.2: Non-EDC and Benign Coverage for our technique, under performance overheads of 10%, 20% and 25%. Lower is better.](image)

Summary: From Figures [6.1] and [6.2] one can observe that under 10% performance overhead, the average EDC coverage is 82%, while the Non-EDC and Benign coverage is about 10%. When the performance overhead is increased to 25%, the average EDC coverage is 86%, while the Non-EDC and Benign coverage is 18%. Using the absolute rates in Table [6.1], and considering the overall EDC causing faults in the applications, this translates to correctly detecting 3.56% from the 4.05% EDC causing faults, while wastefully detecting 10% from the 58% of Non-EDC and benign faults. If
6.3 EDC Coverage under varying funcrange values

we consider the coverage to include all errors except EDC causing ones, this corresponds to an increase in overall coverage from 95.95% without our detectors to 99.5% with our detectors, with 25% performance overhead across the six applications. Therefore, our technique detects EDC causing faults with high coverage, while detecting Non-EDCs and benign faults with low coverage, thereby efficiently differentiating EDC causing faults from the set of all faults in the application.

6.3 EDC Coverage under varying funcrange values

The funcrange is a tunable parameter in the selection algorithm explained in Section 4.3.2. Figure 6.3 shows the effect of funcrange on EDC coverage for the MPEG2 benchmark, for different performance overheads.

![Figure 6.3: Effect of varying funcrange on EDC Coverage, in MPEG2 under performance overheads of 10%, 20% and 25%](image)

With funcrange as 1 (priority to functions with higher execution time), the EDC coverage is 60%, 40%, and 50% under respective performance over-
heads of 10%, 20% and 25%. This is because of the short-sighted nature of the selection algorithm under small window sizes. As the *funcrange* increases to 5, the EDC coverage increases to almost 100%. However, there is almost no benefit in going to values of 7 and 9\(^9\). We have observed a similar effect for other programs, and this is why we use *funcrange* as 5 in our experiments.

### 6.4 EDC coverage under varying fidelity threshold

As mentioned before, for the four Parsec benchmarks, we define EDCs to constitute 30% of the most egregious SDCs. In this section, we consider how the coverage varies if we consider \(X\)% of the most egregious SDCs to be EDCs, where \(X\) varies from 30 to 60. We do not consider JPEG and MPEG2, as they use absolute PSNR values for classifying EDCs.

Figure 6.4 shows the average EDC coverage for the four Parsec benchmarks as the EDC rate increases. As the % of SDCs classified as EDCs increases from 30% to 60%, i.e., the user or application has stricter constraints, the drop in coverage is at most 5% under the given performance overheads. This shows that our algorithm is reasonably robust to changes in fidelity threshold for classifying EDCs. We do not consider threshold values beyond 60% as at such values, EDCs are practically indistinguishable from SDCs (for our benchmarks).

\(^9\)There is a small dip in coverage from window size 5 to 7. We believe this is a statistical anomaly.
6.5 Quantitative comparison with Prior Work

In this section, we quantitatively compare our technique with that of Sundaram et al. [36] who protect an application from soft errors by selective replication. Similar to our technique, they focus on multimedia applications that are a subset of soft-computing applications. However, unlike our approach, they do not distinguish between data that cause large output deviations and those that do not, and hence they protect all pointer and control data. In other words, they do not distinguish between SDC-causing errors and EDC-causing errors.

We implement Sundaram et al.’s technique by considering all control and pointer data as potential EDC data without any ranking or OEF tagging, and use our selection algorithm (with funcrange value of 5) to choose from the EDC data under the given performance bounds. The main difference
6.5. Quantitative comparison with Prior Work

Figure 6.5: EDC coverage by Selective Duplication Technique \cite{36} under performance overheads of 10%, 20% and 25%

with our earlier experiment is the absence of EDC data ranking that selectively detects EDCs from Non-EDCs and benign faults. Figure 6.5 shows the EDC coverage numbers under the given performance overhead bounds. The average EDC coverage is 56.4%, 67.5% and 68.9% under respective performance overheads of 10%, 20% and 25%, which is much lower than our technique (see Figure 6.1), for which the values are 82%, 85% and 86% respectively. The average Non-EDC and benign fault coverage varies from 11% to 17% under the given performance overhead bounds, which is comparable to our technique. Thus, our technique has a higher EDC coverage than that of Sundaram et al. at nearly the same Non-EDC and benign fault coverage.
6.6 Discussion

In this section, we discuss advantages of our detection technique and the threats to validity in our technique.

Our technique identifies locations for placing detectors for EDC causing faults. We compare our placement technique against placing detectors at the end of the program. Detectors placed at the end of the program involves replicating the entire backward slice, starting from the output of the program. This has a much higher performance overhead compared to our technique, which selectively replicates the backward slices of only the chosen detector locations. Further, since the detectors are only placed at the end of the program, recovery using checkpointing would require restarting the program from the start. Using our technique, once an error is detected using our replication based detectors, the program needs to be restarted from the point of deviation, i.e., the start of the backward slice of the detector. Hence, the pre-emptive nature of our detectors takes up lower energy during recovery.

**Threats to Validity:** We estimate the performance overhead as the dynamic count of instructions that are added to the program. This does not consider the effect of the added detectors on the actual running time of the program. The running time might be much larger due to the added code causing cache pollution. However, the total static backward slice is in the order of hundreds when the program code size is in the order of ten thousands, i.e., the static code size increase is around 1%. Hence, the backward slice code getting evicted from the instruction cache is very low.
Also, there are lesser chances of data values of the backward slice code getting evicted from the data cache.

### 6.7 Summary

In this chapter, we presented our results on the coverage of our technique in terms of preemptively and selectively detecting EDC causing faults. The EDC coverage is around 82% under 10% performance overhead, while the Non-EDC and benign coverage is only 10%. From a holistic viewpoint, we improve the baseline coverage (includes all errors except EDC causing ones) by an order of magnitude, from 95.95% to 99.3% under the 10% overhead bound. We also perform a sensitivity analysis with the user having stricter constraints, i.e., more SDCs are classified as EDCs, and the drop in coverage is atmost 5% under given performance overhead bounds. Finally, we compare our heuristics-based static analysis technique against one that randomly chooses detector placement locations from pointer and control data, under the given performance overhead bounds. Our technique detects EDCs with higher coverage of 82% versus 56% under the 10% performance overhead bound.
Chapter 7

Effect of Compiler Optimizations

In this chapter, we study the effect of compiler optimizations on the error resilience of soft computing applications. Compiler optimizations transform the application intermediate code for increased performance, but these transformations can lower error resilience. Our technique for error detector placement incorporates heuristics which are based on static analysis of the application intermediate code. Hence, compiler optimizations have a direct effect on our detector placement technique, and we study the effect of these optimizations.

Recall that resilience is the property of an application to tolerate hardware faults once they have occurred in the application. In the context of soft computing applications, the resilience is the ability of an application to prevent an error from becoming an EDC (see Equation 1.1). Compiler optimizations affect the error resilience of soft computing applications by: (a) modifying the baseline resilience (i.e., without our detection technique) (b) modifying the EDC coverage of our technique. We study the effect of optimizations both with and without our technique. We then classify
these optimizations into resilience packages R1 and R2, based on different levels of resilience (similar to optimization levels of 01 and 02 in optimizing compilers). We study the effect on overall vulnerability by exploring the performance-resilience tradeoff for these optimizations.

7.1 Compiler Optimizations

In this section, we present the specific optimizations under which we evaluate the resilience of soft computing applications, and the rationale behind choosing these optimizations for our study. We choose the optimizations based on the following criteria:

1. Possible dichotomous behaviour on baseline resilience, which makes it difficult to gauge the effect of the optimization

2. Possible change in heuristics, which affects the detector placement locations, and hence the EDC coverage of our technique

3. Possible change in dynamic instructions, making certain regions of code more susceptible to faults.

7.1.1 Dichotomous Behaviour on Baseline Resilience

Faults affecting code transformed by certain optimizations, have conflicting behaviour on the likelihood of an EDC outcome.

Combine Redundant Instructions (Inst-Combine): The Combine Redundant Instructions optimization (Inst-Combine), as the name implies,
7.1. Compiler Optimizations

optimizes code by combining multiple instructions into one. There are two conflicting effects of a hardware fault on the optimized code:

- A hardware fault affecting the combined instruction in the optimized version has a higher likelihood of leading to an EDC compared to a fault affecting the redundant instructions in the unoptimized version. This is because the combined instruction represents the collection of original redundant instructions, and might have a more pronounced effect on the output, thus leading to an EDC.

- The probability of the combined instruction in the optimized version being affected by a fault, is lower than the set of redundant instructions in the unoptimized version, as it has a smaller footprint.

Loop Invariant Code Motion (LICM): Loop invariant code motion \((LICM)\) is an optimization technique which moves invariant code within the loop to the loop pre-header, i.e., it is no longer executed within loop iterations. This optimization is prevalent in popular compilers such as \texttt{gcc}, and LLVM. Note that the effect of the optimization on the code is highly dependent on the application. In particular, it works in applications that contain invariant code within loops that can be moved outside. Similar to Inst-Combine, LICM also brings forth contrasting possibilities in terms of soft errors:

- There is a lower likelihood of a fault affecting the invariant code that is moved to the preheader (executed less frequently than code within the loop), and hence lower likelihood of an EDC.
7.1. Compiler Optimizations

- On the other hand, if a fault affects the invariant code moved outside the loop, there might be a greater propagation of the error since the value is not being reset in each iteration as in the unoptimized version. This leads to a higher likelihood of an EDC.

**Common Sub-Expression Elimination (CSE):** The Common Sub-Expression Elimination optimization (CSE) identifies multiple instances of identical expressions (i.e., they evaluate to the same value), and replaces them with a single variable that holds the computed value. This optimization also brings about dichotomous behaviour of a fault affecting the single variable (similar to the combined instruction in Inst-Combine).

**Global Value Numbering (GVN):** The Global Value Numbering (GVN) optimization eliminates fully and partially redundant code. Note that this optimization might eliminate code that was not found to be redundant in the CSE optimization, and vice versa. This optimization brings about dichotomous behaviour in faults similar to those in Inst-Combine and CSE.

7.1.2 Change in heuristics

Our heuristics are based on branches, pointer data and Optimized EDC Function calls (see Section 4.2). We focus on optimizations that might affect these data, bringing about a change in heuristics.

**Sparse Conditional Constant Propagation (SCCP) and Inter-Procedural SCCP (IP-SCCP):** The Sparse Conditional Constant Propagation optimization (SCCP), propagates constants throughout the code.
7.1. Compiler Optimizations

In the process, certain conditional branches that use these constants become unconditional, which in turn propagates more constants. SCCP is intra-procedural, while Inter-Procedural SCCP (IP-SCCP) does the same optimization across procedures. Also, after SCCP, the static code size is further reduced through dead code elimination (another optimization pass). Since these optimizations remove branches, which is an integral part of our heuristics, we investigate their effect on our EDC coverage.

**Function Inlining (Inline):** Function inlining is an inter-procedural optimization that inlines functions. Note that by this optimization, most optimized EDC functions (OEF) would be inlined. As a result, the heuristic H1 for OEF calls would not be effective, while code within the OEF may now be chosen as detector locations.

**Loop-Unswitch:** This optimization moves branches within loops to outside the loop bodies, and handles the else case of the branch accordingly. This reduces the dynamic number of branches, thereby reducing branch misprediction penalties. Note that these branches moved outside now have larger amount of data within the branch body. Hence a fault affecting this branch has a higher EDC likelihood, leading to a higher EDC rank. However, the chances of a fault striking the branch is lower, since the branch is executed less frequently.

**Loop-Reduce:** This optimization performs strength reduction of array indices within loops. It introduces new add instructions within the loop body to add the index by an appropriate value, and adds phi instruction outside at the loop exit. These instructions might now be better locations to place detectors. Note that in the unoptimized version, these loop indices
were protected as part of loop termination conditions.

**Loop Unrolling (Loop-Unroll):** Loop unrolling (loop-unroll) trades off static code size for performance, by unrolling the loops in a program (as the name suggests). This reduces branch misprediction penalties, as there are fewer branches to (mis-)predict. Also, if the loop statements are independent, higher performance gain can be achieved by loop parallelism. Note that the effect of the loop unrolling is dependent on the loop unrolling factor, and factors such as the loop trip count. This optimization reduces the dynamic count of branch conditions if the loop unrolling is done. Hence, locations other than branches may be better locations to place detectors after unrolling.

**Jump-Threading:** This optimization simplifies control flow by finding distinct threads of control flow running through a basic block. It checks if a conditional branch jumps to a location where another condition is subsumed by the first one. In this case, the first branch is redirected correspondingly, turning two jumps into one. Since this optimization modifies control flow by affecting branches, the placement locations for branches might change.

### 7.1.3 Change in EDC-prone code regions

These optimizations change the dynamic code size, possibly making other regions of code (than the ones chosen by our algorithm) more susceptible to EDC causing faults.

**Aggresive Dead Code Elimination (ADCE):** As the name suggests, this optimization removes dead code from the application.

**Loop Simplification (LoopInstSimplify):** The LoopInstSimplify op-
7.2 Methodology

Optimization simplifies the code within loops, by canonicalizing natural loops. LoopInstSimplify guarantees preheader and exit blocks for each loop. It modifies the Control Flow Graph.

Merge-return: As the name implies, Merge-return unifies all the exit blocks in the function, and ensures that functions have at most one return instruction in them.

7.2 Methodology

We evaluate our technique by performing fault injection on the benchmark programs in Table 5.1 using LLFI. These fault injection experiments are performed on fifteen versions of each benchmark, namely the unoptimized code, and the fourteen optimized versions. For now, we consider only one optimization at a time. All the experiments are run with the O0 option. We keep the fidelity threshold value and number of injections the same across all versions.

We measure the EDC coverage of our technique under the fifteen versions of the six benchmarks. We injected 5000 faults for each benchmark and optimization combination. The EDC rates are statistically significant within an error bar of ±0.8% at the 95% confidence interval. We measure the EDC coverage under the performance overhead bound of 20% (the dynamic instruction count of the replicated instructions), for all versions.

Analysis approach: We study the effect of the compiler optimizations on the baseline resilience by analyzing how the EDC rates vary between the unoptimized and optimized version. When the EDC rates are within
the error bars, they are considered to be the same between optimized and unoptimized. When the rates vary beyond the error bars, we perform the two-sample hypothesis test (z test) to determine if they are different. The null and alternative hypotheses are as follows:

1. H0: The EDC rates between the unoptimized and optimized versions are the same.

2. H1: The EDC rates between the unoptimized and optimized versions are different.

We favour the alternative hypothesis over the null hypothesis, i.e., we consider the EDC rates to be different, when the p-value is less than 0.1 (we measure both the 0.05 and 0.1 levels).

We study the effect of the optimizations on our technique by analyzing the change in EDC coverage. We consider an optimization safe under our technique, if the EDC coverage is comparable to that of the unoptimized code, i.e., higher or within 5% lower with respect to the unoptimized code. In cases where the optimization is unsafe, we qualitatively examine the detector locations chosen in the optimized version to understand why the EDC coverage is lower.

**Resilience Packages:** Based on the results of the analysis of safe optimizations, we group the compiler optimizations into resilience packages. This helps us identify which optimizations can be used under varying requirements of resilience. R1 represents the optimizations offering the highest resilience i.e., they are safe for all benchmarks. R2 represents optimizations that offer the next highest resilience, i.e., they are safe for most benchmarks.
(five out of six).

7.3 Results

In this section, we present the EDC rates, i.e., the effect of optimizations on the baseline resilience. We then present the effect of the optimizations on our detection technique, by reporting the EDC coverage of our technique for the fourteen optimizations under the 20% performance overhead.

7.3.1 Effect of compiler optimizations on Baseline Resilience

Table 7.1 shows the EDC percentages across the benchmarks for the fourteen optimizations along with the unoptimized version. The last row of the table is the error bars of the EDC percentages for the respective benchmarks. For cases where the optimizations varied from the unoptimized version (beyond the error bars), we indicate the direction of the variation and the p-value levels (0.1 or 0.05).

The optimization reduces the error resilience of the application compared to baseline resilience, when the EDC rate of the optimized version of the application is higher compared to the unoptimized one. The effect of optimizations on the baseline resilience depends on benchmark specific characteristics. For example, JPEG and MPEG are the only programs that maintain the baseline resilience across all the optimizations.
### Table 7.1: Percentage of EDC outcomes in each benchmark under the fourteen optimizations, and the unoptimized version. In cases where rate is beyond error bars, we add H or L (signifying higher or lower). * signifies p-value $\leq 0.05$ and ** signifies p-value $\leq 0.1$

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Black-scholes</th>
<th>X264</th>
<th>Canneal</th>
<th>Swap-tions</th>
<th>JPEG</th>
<th>MPEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst-Combine</td>
<td>9.9</td>
<td>2.48</td>
<td>4.56</td>
<td>H*</td>
<td>2.5</td>
<td>3.68</td>
</tr>
<tr>
<td>LICM</td>
<td>9.48</td>
<td>2.96</td>
<td>3.26</td>
<td>H**</td>
<td>3.12</td>
<td>3.56</td>
</tr>
<tr>
<td>SCCP</td>
<td>9.38</td>
<td>2.1</td>
<td>3.94</td>
<td>H</td>
<td>2.98</td>
<td>4.08</td>
</tr>
<tr>
<td>IP-SCCP</td>
<td>10.48</td>
<td>2.28</td>
<td>3.92</td>
<td>H</td>
<td>2.98</td>
<td>4.22</td>
</tr>
<tr>
<td>Inline</td>
<td>8.68</td>
<td>2.3</td>
<td>3.86</td>
<td>H</td>
<td>2.44</td>
<td>3.86</td>
</tr>
<tr>
<td>Loop-Unswitch</td>
<td>10.36</td>
<td>2.34</td>
<td>3.22</td>
<td>2.32</td>
<td>3.28</td>
<td>3.66</td>
</tr>
<tr>
<td>Loop-Reduce</td>
<td>9.14</td>
<td>2.7</td>
<td>3.28</td>
<td>2.78</td>
<td>3.0</td>
<td>3.82</td>
</tr>
<tr>
<td>Loop-Unroll</td>
<td>8.48</td>
<td>2.82</td>
<td>3.26</td>
<td>2.44</td>
<td>4.16</td>
<td>2.56</td>
</tr>
<tr>
<td>Jump-Threading</td>
<td>10.92</td>
<td>2.7</td>
<td>3.32</td>
<td>2.76</td>
<td>3.8</td>
<td>2.38</td>
</tr>
</tbody>
</table>
7.3. Results

<table>
<thead>
<tr>
<th>Optimization</th>
<th>EDC</th>
<th>H</th>
<th>H*</th>
<th>H</th>
<th>H**</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCE</td>
<td>9.28</td>
<td>2.22</td>
<td>3.98</td>
<td>2.72</td>
<td>3.82</td>
</tr>
<tr>
<td>CSE</td>
<td>13.94</td>
<td>2.02</td>
<td>4.04</td>
<td>2.76</td>
<td>3.7</td>
</tr>
<tr>
<td>Loop-InstSimplify</td>
<td>9.1</td>
<td>2.4</td>
<td>3.18</td>
<td>2.24</td>
<td>3.6</td>
</tr>
<tr>
<td>Merge-return</td>
<td>11.4</td>
<td>2.52</td>
<td>3.04</td>
<td>2.58</td>
<td>4.12</td>
</tr>
<tr>
<td>GVN</td>
<td>11.02</td>
<td>2.3</td>
<td>3.56</td>
<td>3.2</td>
<td>3.66</td>
</tr>
<tr>
<td>UnOpt</td>
<td>10</td>
<td>2.24</td>
<td>3.32</td>
<td>2.36</td>
<td>3.76</td>
</tr>
<tr>
<td>Error Bars</td>
<td>0.8</td>
<td>0.4</td>
<td>0.49</td>
<td>0.42</td>
<td>0.53</td>
</tr>
</tbody>
</table>

From Table 7.1, the optimizations Inline, Loop-Unswitch and Loop-InstSimplify strictly maintain the baseline resilience, i.e., their EDC rates are within the error bars, compared to the unoptimized versions. We investigate the optimizations that significantly lower the baseline resilience, based on the p-values\(^\text{10}\).

**Inst-Combine:** In general, Inst-Combine optimized benchmarks have a lower EDC rate than the corresponding unoptimized version, thereby maintaining the baseline error resilience of the application. However, Inst-Combine optimized Canneal has a higher EDC rate than its unoptimized version. On further investigation, we found that the dynamic count of in-

\(^{10}\)Note that a lower p-value implies that we can reject H0 (the null hypothesis) with more certainty.
7.3. Results

Instructions in the optimized version is around 25% lower than the unoptimized version. For all other benchmarks, the difference in dynamic count is within 1%. Hence, faults affecting Inst-combine optimized code have a higher likelihood of causing an EDC, if and only if the reduction in dynamic code size with respect to the unoptimized version is highly pronounced.

LICM: As shown in Table 7.1, all benchmarks except Swaptions and X264 have LICM optimized EDC rate similar to that of the unoptimized version. Swaptions has a higher EDC rate than the baseline unoptimized version, i.e., the optimization lowers its resilience compared to the unoptimized version. Swaptions contains many instances of invariant code within loops. Figure 7.1 shows two such functions, Discount_Factors_Blocking and HJM_SimPath_Foward_Blocking. The loop invariant code in Discount_Factors_Blocking is \( i \times \text{BLOCKSIZE} + b \) at line 6, which is hoisted outside the loop. Similarly, for HJM_SimPath_Foward_Blocking, we have multiple instances of invariant code such as \( iN-1 \) at line 15 and 18, and \( \text{BLOCKSIZE} \times j + b \) at line 19. In the LICM optimized version for the HJM_SimPath_Foward_Blocking function, we did not see faults affecting the invariant code at all.

In the unoptimized version, faults affect multiple invariants within the loop (the probability of the fault striking is much higher as the invariant code is within the loop), and many of them lead to benign outcomes or Non-EDCs.

It is interesting to note that the increase in EDC rate for Swaptions compared to its unoptimized version, is not due to faults affecting the invariant code outside the loop (we did not observe faults affecting the invariant code). The increase is due to faults affecting loop termination conditions whose loop body contains a call to the function \texttt{RanUnif()}, and these faults
7.3. Results

Figure 7.1: Loop Invariant Code Examples in Swaptions

lead to EDCs. These loop bodies contain invariant code similar to those in function `HJM_SimPath_Forward_Blocking`. Due to the reduction in dynamic code size of statements within the loop body of the LICM-optimized Swaptions, there are a higher number of faults affecting the loop termination conditions, and hence a higher number of EDCs. Also, we found that if a random hardware fault occurs in the application, there is a low likelihood of the fault affecting invariant code hoisted outside the loop in LICM optimized code.

**CSE, Merge-return and GVN:** In all three optimizations, the lowering in baseline resilience occurs in benchmarks that have a highly pronounced dynamic code size reduction. The Global Value Numbering (GVN) optimization has a lowered baseline resilience for Swaptions, which has around 19% reduction in dynamic code size. CSE and Merge-return also have lowered resilience for Blackscholes, which has a large reduction in dynamic code
size, compared to other benchmarks.

**Summary:** In all optimizations except LICM, benchmarks that have a significant reduction in baseline resilience (i.e., their EDC rate is higher than the baseline EDC rate) also have a large reduction in dynamic code size compared to other benchmarks. Intuitively, a large reduction in dynamic code size implies that a fault affecting the application has a more pronounced effect, i.e., higher likelihood of causing an EDC.

However, this is not the case for the LICM optimization. Both Swaptions and X264 LICM-optimized code have a reduction in baseline resilience. While Swaptions has a significant decrease in dynamic code size, X264 does not have a decrease. X264 is an anamolous case where the static transformations do not affect the dynamic code size. The reason for reduction in baseline resilience is due to higher number of faults affecting the code regions within the function `cabac_encode_decision_c`, which lead to EDCs.

### 7.3.2 Effect of compiler optimizations on detection technique

We present the effect of the fourteen optimizations on the EDC coverage of our technique, and which optimizations are safe under our detection technique (beyond the change in baseline resilience). Figure 7.3 shows the normalized EDC coverage with respect to the baseline coverage for the fourteen optimizations. The baseline EDC coverage is the coverage for the unoptimized version of the benchmark. We categorize the optimizations into resilience packages R1 and R2 as follows:

**Resilience Package R1:** This resilience package contains the opti-
7.3. Results

mizations that provide the highest degree of resilience. These optimizations are *safe* under our technique for all benchmarks, i.e., the coverage is higher or within 5% lower than the baseline coverage. From Figures 7.3(b), 7.3(e), 7.3(j), and 7.3(k), LICM, function inline, ADCE and CSE are safe optimizations for all benchmarks. Hence, we include these *four optimizations in the R1 resilience package*.

**Resilience Package R2:** This resilience package contains optimizations that provide a lower degree of resilience. These optimizations are safe for five out of six benchmarks. *The R2 resilience package includes six optimizations* namely, IP-SCCP, Loop-Unswitch, Jump-Threading, Loop-InstSimplify, Merge-Return and GVN.

From Figures 7.3(d), 7.3(f) and 7.3(i), IP-SCCP, Loop-Unswitch and Jump-Threading are safe for all benchmarks except Blackscholes. These three optimizations insert loop pre-headers, and an extra branch before the loop. In function `bs_thread` of Blackscholes (see Figure 7.2), a large number of faults affect the loop termination condition at line 3. This leads to EDCs, both in the unoptimized and the optimized version. In the optimized version, a new branch condition is added to the loop preheader, and this condition is chosen for detector placement instead of the loop termination condition. This new location chosen in the optimized code does not contribute to the EDC coverage. Further, in the original code, the loop termination condition is chosen by our algorithm, and it detects the EDCs caused at that location.

Optimizations Loop-InstSimplify, Merge-Return and GVN are safe for all benchmarks except Swaptions. In Merge-Return, although there is no
7.3. Results

Figure 7.2: Blackscholes code segment

change in the dynamic code size compared to Swaptions unoptimized, higher number of faults affect the code region in HJM_Swaptions Blocking function of Swaptions. This function performs the main calculation of the Monte-Carlo simulations (10% of total EDC in mergereturn optimized versus 5% in unoptimized). In GVN and Loop-InstSimplify, there are higher number of faults affecting the RanUnif() function of Swaptions, compared to the unoptimized code.

Loop-Reduce has a high increase in resilience with our technique, which is due to the Swaptions benchmark (see Figure 7.3(g)). Our technique protects a particular branch instruction which increases the EDC coverage to 100%. However, protecting this branch instruction increases the benign coverage to around 85%. This destroys the selectivity of our technique, i.e., detecting Non-EDCs and benign faults with low coverage.

Summary: From Sections 7.3.1 and 7.3.2 we find that maintaining a resilience guarantee under compiler optimizations is entirely dependent on which regions of code become more susceptible to EDC causing faults. For example, in cases such as LICM-optimized Swaptions and Inst-Combine optimized Canneal, the baseline resilience of the optimized version is lower.
7.3. Results

(a) Normalized EDC Coverage of Inst-Combine

(b) Normalized EDC Coverage of LICM

(c) Normalized EDC Coverage of SCCP

(d) Normalized EDC Coverage of IP-SCCP

(e) Normalized EDC Coverage of Inline

(f) Normalized EDC Coverage of Loop-Unswitch
7.3. Results

(g) Normalized EDC Coverage of Loop-Reduce
(h) Normalized EDC Coverage of Loop-Unroll

(i) Normalized EDC Coverage of jump-threading
(j) Normalized EDC Coverage of ADCE

(k) Normalized EDC Coverage of CSE
(l) Normalized EDC Coverage of LoopInst-Simplify
Figure 7.3: Normalized EDC Coverage of different optimizations with respect to the Unopt EDC Coverage (shown as 100%).

compared to its unoptimized version (in the absence of our detection technique), i.e., the EDC rate is higher compared to the unoptimized version. However, our detection technique maintains the overall resilience of these applications, because the increased number of EDC causing faults in the optimized version affects locations that are already protected by our technique. In other cases such as Inst-Combine optimized X264 and Swaptions, the baseline resilience is maintained in the optimized version (the EDC rate is lower or similar), but these lower number of EDC causing faults affect locations not protected by our technique.

7.4 Overall Vulnerability

In this section, we study the effect of the fourteen optimizations on the overall vulnerability of soft computing applications by investigating the tradeoff between performance and the error resilience. Recall that vulnerability for
soft computing applications is the probability of a fault affecting an application and that fault leading to an EDC (see Equation 1.2), and can be calculated as the product of the EDC rate and the execution time.

For example, when the EDC rate is higher in some cases compared to the unoptimized version, the overall vulnerability might be lower than the unoptimized version, if the execution time is lower. This is because there is a lower probability of faults affecting the application. Conversely, when the EDC rate is low, but the execution time is high, the vulnerability might be higher as there is a higher probability of faults affecting the application.

Compiler optimizations generally improve performance of applications by transforming the code. However, some optimizations might have a detrimental effect on the resilience of the application. For example, the Inst-Combine optimization improves performance by combining redundant instructions and reducing the dynamic code size. However, the resilience is lowered because faults affecting this reduced code size have a more pronounced effect leading to higher number of EDCs (see Section 7.3.1).

We explore this performance-resilience tradeoff for the fourteen optimizations averaged across the six benchmarks. This provides us with a holistic view of each individual optimization, i.e., how the optimization affects the vulnerability of the application.

We assume a hardware model where all instructions take one cycle per instruction (CPI), which makes the execution time directly proportional to the dynamic instruction count. We study the relative effect of an optimization on the vulnerability of the application in three cases:
7.4. Overall Vulnerability

C1. the vulnerability with and without our detectors under each optimization

C2. the baseline vulnerability (i.e., without our detectors) in comparison to that of the unoptimized version across optimizations

C3. the vulnerability of the code protected with our detectors in comparison to that of the unoptimized version across optimizations

Table 7.2 shows the vulnerability (in terms of the product of dynamic instruction count and the EDC rate), for each of the fourteen optimizations and the unoptimized code, averaged across the six benchmarks. Column 4 shows the baseline vulnerability, while column 7 shows the average vulnerability of applications protected by our detectors, i.e., the product of dynamic instruction count and the undetected EDC rate. The undetected EDC rate is the fraction of undetected EDCs out of the total set of injected faults. Note that a lower vulnerability is better.

Baseline vulnerability versus vulnerability with detection technique (C1): Column 8 in Table 7.2 (ratio) shows the reduction in vulnerability of our technique compared to baseline vulnerability. The vulnerability of applications with our detectors (column 7) is lower than the baseline vulnerability (column 4) for all optimizations. Twelve optimizations out of fourteen have at least a 2X reduction in vulnerability with our technique. This is because even though the dynamic count increases by 20% with our detectors, the reduction in undetected EDC rate offsets this increase by a much higher extent. Note that Loop-Reduce is an anomalous case as explained in Section 7.3.2.
### 7.4. Overall Vulnerability

Table 7.2: The overall vulnerability of the fourteen optimizations both with and without our technique (averaged across the six benchmarks) The lower the vulnerability, the better the optimization. The highlighted optimizations have reduced vulnerability compared to unoptimized version.

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Baseline</th>
<th>With our Technique</th>
<th>Ratio (x/y)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in mil-</td>
<td>Rate (E) (x = IB * E)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(IB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst-Combine</td>
<td>215.87</td>
<td>4.19</td>
<td>552.59</td>
</tr>
<tr>
<td>LICM</td>
<td>180.37</td>
<td>4.08</td>
<td>558.66</td>
</tr>
<tr>
<td>SCCP</td>
<td>216.86</td>
<td>4.03</td>
<td>636.21</td>
</tr>
<tr>
<td>IP-SCCP</td>
<td>215.18</td>
<td>4.32</td>
<td>641.03</td>
</tr>
<tr>
<td>Inline</td>
<td>197.98</td>
<td>3.81</td>
<td>487.59</td>
</tr>
<tr>
<td>LoopUnswitch</td>
<td>179.45</td>
<td>4.13</td>
<td>576.38</td>
</tr>
<tr>
<td>Loop-Reduce</td>
<td>184.58</td>
<td>4.02</td>
<td>556.4</td>
</tr>
<tr>
<td>Loop-Unroll</td>
<td>201.79</td>
<td>3.95</td>
<td>516.22</td>
</tr>
</tbody>
</table>
Baseline vulnerability analysis (C2): When comparing the baseline vulnerabilities across optimizations (column 4 in Table 7.2), the optimizations Inline, Loop-Unroll, CSE, and Loop-InstSimplify have a lower vulnerability compared to the unoptimized version. CSE has a much higher EDC rate compared to the unoptimized version, but the reduction in dynamic code size (compared to the unoptimized code) offsets this higher EDC rate. Hence, the probability of a fault affecting the application is lower than the unoptimized version, which in turn makes the vulnerability lower than the unoptimized version.

Vulnerability analysis with our detectors (C3): From column 7 in Table 7.2 the optimizations Inline, Loop-reduce, CSE and Loop-InstSimplify have lower average vulnerability than the unoptimized version. Loop-InstSimplify has a lower resilience with our technique, i.e., it belongs to the resilience

<table>
<thead>
<tr>
<th></th>
<th>Jump-Threading</th>
<th>ADCE</th>
<th>CSE</th>
<th>LoopInst-Simplify</th>
<th>Merge-return</th>
<th>GVN</th>
<th>UnOpt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>212.59</td>
<td>201.19</td>
<td>180.02</td>
<td>200.63</td>
<td>216.29</td>
<td>179.68</td>
<td>217.97</td>
</tr>
<tr>
<td></td>
<td>4.31</td>
<td>4.03</td>
<td>4.74</td>
<td>3.74</td>
<td>4.27</td>
<td>4.31</td>
<td>3.99</td>
</tr>
<tr>
<td></td>
<td>601.49</td>
<td>554.9</td>
<td>511.36</td>
<td>464.88</td>
<td>569.33</td>
<td>573.45</td>
<td>538.68</td>
</tr>
<tr>
<td></td>
<td>255.11</td>
<td>241.43</td>
<td>216.03</td>
<td>240.76</td>
<td>259.55</td>
<td>215.62</td>
<td>261.56</td>
</tr>
<tr>
<td></td>
<td>0.82</td>
<td>0.67</td>
<td>0.76</td>
<td>0.68</td>
<td>0.73</td>
<td>0.81</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>257.64</td>
<td>253.58</td>
<td>200.73</td>
<td>223.16</td>
<td>304.4</td>
<td>283.89</td>
<td>231.62</td>
</tr>
<tr>
<td></td>
<td>2.33</td>
<td>2.19</td>
<td>2.55</td>
<td>2.08</td>
<td>1.87</td>
<td>2.02</td>
<td>2.32</td>
</tr>
</tbody>
</table>
package R2, but it performs better in terms of vulnerability.

7.5 Summary

In this chapter, we study the effect of fourteen compiler optimizations on the error resilience of soft computing applications. Based on the results of our fault injection experiments, we group the optimizations into resilience packages to identify optimizations that guarantee varying degrees of resilience. Our technique maintains a reasonable resilience guarantee for ten out of fourteen optimizations. In the remaining four optimizations, other code regions become more susceptible to EDC causing faults, and these regions are not protected by our technique. Further, the effect of the optimizations is highly dependent on benchmark specific characteristics. We also study the overall vulnerability of soft computing applications, and we find that there is a large decrease in vulnerability of the applications protected by our detectors compared to their baseline vulnerability (i.e., without our technique), for all the optimizations.
Chapter 8

Conclusion and Future Work

8.1 Conclusion

Soft computing applications tolerate most errors that result in deviations in output or Silent Data Corruptions (SDCs). However, they do not tolerate outcomes that deviate significantly from the fault-free outcome, e.g. major glitches in decoded video. We classify such outcomes as Egregious Data Corruptions (EDCs).

This thesis presents an automated technique for detector placement in soft computing applications for pre-emptively and selectively detecting EDC causing errors. We first perform a fault injection study to identify the characteristics of these errors and formulate heuristics for identifying EDC-prone code regions. Our initial study showed that the EDC causing errors constitute only a small percentage of non-crashing errors, and hence it is crucial to selectively detect these errors from the set of benign and SDC errors. We develop an automated algorithm based on static analysis and runtime profiling, for identifying detector locations for detecting EDC causing errors with high coverage, bounded by a given performance overhead. We find that the detectors placed by the algorithm achieve an average EDC coverage of 82% under 10% performance overhead, while detecting only 10% of
8.2. Future Work

We can extend our work on error detector placement in soft computing applications, to enhance the reliability of these applications as a whole.

First, we plan to implement the actual detectors and evaluate their coverage. In our work, we consider ideal detectors and focus on the problem of detector placement, rather than detector derivation. Our detectors are based on backward slice replication.

Second, with increasing error rates and energy consumption being of prime importance, focusing on error recovery of these applications is another direction for future work. Similar to detection, this problem also has two dimensions: (1) placement of recovery code for optimum energy and performance and (2) the semantics of the recovery code. Although, the locations chosen for recovery might be different from our detector placement locations, our initial fault injection experiments and heuristics may be useful in developing the semantics of the recovery code.

Third, we plan to identify different combination of optimizations that maintain or lower the overall vulnerability of soft computing applications. Programmers generally apply optimizations collections (such as O2 or O3 in gcc) to aggressively improve the performance of applications. In our current...
work, we investigate the effect of individual compiler optimizations on the error resilience of these applications. This has the disadvantage that the effect of the collective combination of optimizations may be different from individual optimizations, and may not work for all applications. Modern compilers such as \texttt{gcc} use heuristics to decide which groups of optimizations should be applied to applications (in terms of the order of the optimization and the number of optimizations). There has also been some work on employing machine learning to choose these optimizations for performance improvement. Similar techniques could be employed to choose the set of optimizations that guarantees a high performance-resilience product, i.e., lower vulnerability for soft computing applications.
Bibliography


[29] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. MICRO 36, pages 29–, 2003.


[31] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by dupli-


[38] Anna Thomas, Jacques Clapauch, and Karthik Pattabiraman. Effect
of compiler optimizations on the error resilience of soft computing applications. AER, 2013.


